Développement d'un environnement de simulation de systèmes complexes. Application aux bâtiments

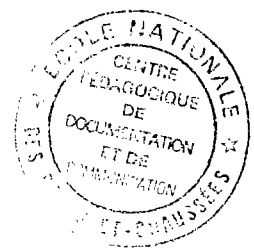
Rolf Ebert

THÈSE DE DOCTORAT

Spécialité : Sciences et Techniques du Bâtiment

Auteur : **Rolf Ebert**

Développement d'un environnement de simulation de systèmes complexes application aux bâtiments



soutenue le **25. Novembre 1993** devant le jury composé de MM. :

J. Halin	Rapporteur
J.-P. Grandpeix	Rapporteur
R. Lalement	Examinateur
G. Lefebvre	Directeur de thèse
J. Rilling	Examinateur
M. Riveill	Examinateur

Résumé

Ce travail présente l'application de concepts modernes du génie logiciel au développement d'un environnement de simulation modulaire de systèmes thermiques complexes.

Une application stricte des idées de la *Conception Orientée Objet* permet la réalisation d'un environnement modulaire, souple et efficace. Une analyse hiérarchique du système à étudier nous fournit un graphe de dépendance entre les composants du système. Chaque composant est confiné dans un objet informatique et a son propre modèle de calcul. Le travail essentiel du simulateur est la résolution des conditions de connexion entre les composants. La structure hiérarchique est conservée et exploitée pour une parallélisation des calculs pendant la simulation numérique. Pour une meilleure capitalisation des modèles développés, nous nous reposons sur une « *modélothèque* » à partir de laquelle l'utilisateur peut récupérer des modèles pour construire son propre système de simulation.

Des exemples d'application de notre environnement de simulation pour des problèmes thermiques de bâtiment sont présentés.

Abstract

This work presents the application of modern software engineering concepts on the development of a modular simulation environement for the simulation of complex thermal systems.

A strict application of the ideas of *object oriented design* as limited access to internal data allows the realisation of a modular, flexible and effective simulation environment. A hierarchical analysis of the studied system gives us a graph describing the dependencies beteween the components of the system. Each component is encapsulated in a computer object with its own model of calculation. The actual work of the simulation program is the resolution of the connection conditions. The hierarchical structure stays valid during the numerical resolution and is used for parallelization of the necessary calculations. In order to ease the usage of once developed models we base the environment on a model library from which the user can pick the models and assemble his/her own simulation system.

Some examples show the the application of our simulation environment to thermal problems of building technology.

Zusammenfassung

Diese Arbeit zeigt die Anwendung moderner Techniken der Softwareentwicklung anhand einer modularen Simulationsumgebung zur Simulation komplexer, thermischer Systeme.

Eine strikte Anwendung der Ideen des *objekt-orientierten Designs* wie der begrenzte Zugang zu internen Daten der Modelle ermöglicht die Realisierung einer modularen, flexiblen und effiziente Simulationsumgebung. Eine hierarchische Analyse des zu untersuchenden Systems liefert uns einen Graphen der die Abhängigkeiten zwischen den Systemkomponenten beschreibt. Jeder Baustein ist in ein Computer-Objekt dargestellt und hat hat sein eigenes Rechenmodell. Die eigentliche Arbeit des Simulationsprogrammes ist die Auflösung der Verbindungsbedingungen zwischen den Komponenten. Die hierarchische Struktur bleibt auch während der Auflösung erhalten und wird für eine Parallelisierung der Berechnungen ausgenützt. Um eine bessere Ausnutzung einmal entwickelter Modelle zu erleichtern wurde eine

„Modelothek“ geschaffen. Ein Anwender kann aus dieser beliebig seine Modelle entnehmen und damit sein eigenes Simulationssystem zusammenstellen.

Beispiele zeigen die Anwendung unserer Simulationsumgebung auf thermische Probleme aus dem Bereich der Gebäudetechnik.

Remerciements

Le travail de recherche décrit dans ce mémoire a été effectué au sein du « *Groupe Informatique et Systèmes Énergétiques*», un groupe de recherche commun à l'Ecole des Mines de Paris et l'École Nationale des Ponts et Chaussées. Je tiens à exprimer ici le plaisir d'avoir préparé ma thèse dans cette équipe.

En cette fin de travail, mes remerciements vont :

- à Mr. RILLING, responsable de formation doctorale à l'ENPC et directeur scientifique au CSTB, qui a accepté de présider mon jury de thèse, et a soutenu ma demande d'obtention de bourse auprès de l'ENPC,
- à Mr. J.-Y. GRANDPEIX, Docteur d'état et responsable de recherche au CNRS, pour l'intérêt qu'il a réservé à mes travaux et pour son acceptation d'être rapporteur de ma thèse,
- à Mr. J. HALIN, Professeur à l'Ecole Fédérale Polytechnique de Zurich, pour accepter la fonction de rapporteur,
- à Mr. R. LALEMAND, Professeur à l'École Nationale des Ponts et Chaussées, pour son soutien et sa participation au jury de thèse,
- à Mr. M. RIVEILL, Professeur à l'Université de Chambéry, pour son intérêt à mon travail et sa participation au jury de thèse,
- à Mr. G. LEFEBVRE, responsable du GISE, pour m'avoir accueilli dans ce groupe et pour son soutien et ses conseils au cours de cette thèse. Il m'a apporté de bonnes conditions financières et de travail qui m'ont permis de m'intégrer facilement dans une ville, un pays et une langue etrangère.
- à l'« *Agence de l'Environnement et de la Maîtrise de l'Énergie* » pour le financement matériel de cette recherche,
- à Mr. J.-M. NATAF pour ses conseils scientifiques et linguistiques précieux et son soutien amical,
- aux membres du laboratoire, particulièrement H. OULEFKI, mais également E. WURTZ, C. DEVILLE-CAVELLIN, B. FLAMENT, K. ELKHOURY, Y. BONNEFOUS, A. NEVEU, T. SALSET, R. KERIVEN, et G. CAPLAIN pour leur aide amicale,
- et enfin à Veronika LANGER qui m'a continuellement soutenu et encouragé dans l'accomplissement de cette thèse.

Sommaire

Chapitre 1

Introduction

Dans beaucoup de laboratoires qui se consacrent à la modélisation de systèmes thermiques on a pu constater jusqu'à une époque récente le développement d'outils informatiques dédiés à des systèmes et/ou méthodes spécifiques. Ces outils ont souvent de nombreux points communs au niveau théorique. Mais au niveau informatique ces logiciels sont maintenus par des personnes différentes et leur communication, c'est à dire, l'échange de données entre les outils au niveau de la machine, est pratiquement impossible. Pour chaque nouveau système thermique et pour chaque nouvelle modélisation, on est forcé de concevoir une nouvelle base de données, de nouveaux algorithmes et éventuellement réécrire un programme entier. Ce problème a causé le développement de nouveaux environnements logiciels un peu partout dans le monde. Par ailleurs, on peut observer ces dernières années une évolution rapide des paradigmes de programmation souvent désignés par le mot-clef *programmation orientée objet.* Créé pour répondre à la nécessité de surmonter les problèmes de la modularité, de la réutilisabilité et de la maintenance et profitant des nouveaux moyens informatique, le projet SYMBOL fut fondé dans le groupe GISE.

Nous allons présenter un outil de simulation qui se situe à l'intérieur de l'environnement logiciel de modélisation, SYMBOL. Bien que ses applications potentielles soient plus larges, il est actuellement utilisé à la modélisation des transferts thermiques dans les systèmes complexes tels que les bâtiments. L'objectif principal du projet global est l'amélioration des connaissances en matière de systèmes thermiques par la modélisation et l'analyse. Dans ce but sont développés un ensemble d'outils logiciels de modélisation. Une des phases principales d'une étude de système est la simulation de l'ensemble du problème. Il en résulte la nécessité de disposer d'un logiciel d'analyse quantitative. Dans le cadre du projet SYMBOL qui se veut plus vaste nous allons donc présenter le logiciel Motor-2. C'est l'unique programme dans SYMBOL qui permette de calculer le comportement non-linéaire dans le temps d'un ensemble d'objets raccordés.

Ce logiciel et les idées sous-jacentes ne sont pas limités à la simulation des problèmes thermiques du bâtiment. Comme nous partons d'une approche modulaire, la création de bibliothèques de modèles est facile. Ces bibliothèques peuvent être classées par domaine d'application. Dans le cadre de ce travail,

nous nous sommes surtout appliqués à produire des exemples relevant de la thermique du bâtiment.

Le noyau de simulation est le logiciel Motor-2. Dans sa conception, il est basé sur un logiciel plus ancien qui a été développé dans le cadre d'un stage scientifique (voir le logiciel Motor dans [30]). Le nom du logiciel est également hérité de ce travail. À l'époque, nous créions des morceaux de code de programmation, deux parties pour chaque composant du système, d'où le nom *MO*del genera*TOR*.

Le plan de ce rapport fait apparaître globalement cinq parties. Dans la première partie, le chapitre 2, nous développons d'abord le cadre général de la simulation et de la modélisation. Une présentation de quelques environnements de simulation existant, ciblés à la thermique de bâtiment, termine ce chapitre. La deuxième partie qui comprend les chapitre 3 et 4, donne une première vue des objectifs et principes de base du logiciel Motor-2. Les deux chapitres suivants (5 et 6) décrivent en détail les concepts de la modularité et l'approche orientée objet tels qu'ils sont appliqués à notre environnement de simulation. Les stratégies algorithmiques qui résultent de ces concepts s'y trouvent également. Une quatrième partie (chap. 7), plus petite, présente l'environnement qui entoure le programme de simulation. Les derniers deux chapitres 8 et 9 illustrent l'application de Motor-2 à l'aide de quelques exemples. La difficulté d'un bon choix de découpage y est étudiée.

Chapitre 2

Contexte et état de l'art de la simulation

Les systèmes étudiés de nos jours deviennent plus grands et complexes, ce qui les rend difficiles à comprendre. Une analyse structurée du système peut nous amener à une description pertinente pour la simulation du système.

Abordons le problème de la complexité. Nous allons d'abord présenter une approche générale pour aborder ce genre de problème, souvent connue comme *approche systémique*. Ensuite les questions relatives à la simulation seront traitées. Toutes les stratégies adoptées par les divers laboratoires afin de réduire la complexité d'un système thermique utilisent la même idée de base : la modularité. Il n'est pas étonnant de trouver des approches différentes pour s'attaquer à ce problème. Un recensement des environnements de simulation actuellement disponibles peut nous donner une idée de l'état actuel de l'avancement des travaux. Nos propres idées pour un environnement de simulation terminent ce chapitre.

2.1 Contexte

2.1.1 Étude de systèmes : modélisation et simulation

Le mot *modélisation* revêt des significations très différentes selon les interlocuteurs :

- pour certains la modélisation correspond à l'ensemble des activités qui permettent la création, la mise au point et l'exécution sur un ordinateur de maquettes virtuelles des systèmes à étudier.

- pour d'autres, la modélisation se confond avec la *simulation numérique*, c'est à dire avec la résolution d'équations d'évolution de la physique.

- pour d'autres enfin, il s'agit de l'*élaboration de relations* entre les variables caractéristiques d'un système ou processus physique donné, capables de bien simuler le comportement de ce système dans un contexte donné.

Notre point de vue correspond à la dernière définition. Elle est certainement plus restreinte que la première, mais dans l'étude d'un système nous voulons distinguer ici deux phases différentes : le développement de modèles d'une part et leur utilisation dans une étude concrète d'autre part. Seule la première phase sera appelée la *modélisation* et alors que pour la deuxième phase, nous étendrons le sens du mot *simulation*. L'exploitation de modèles n'est certainement pas limitée à la simulation. La simulation est un des objectifs possibles de la modélisation ; mais c'est ce qui nous intéresse ici. Cette séparation en deux phases a comme objectif la séparation possible des travaux d'un développeur de modèles de ceux d'un ingénieur de la simulation (utilisateur de modèles). La distinction entre les deux peut contribuer à l'échange des modèles et à leur application.

La première phase est la modélisation. Cette démarche essaye d'abstraire le monde réel pour trouver une image pertinente de la réalité. Cette création d'un concept se fait par des exemples et des généralisations[1]. En fait, chaque objet du monde réel n'a pas de définition *per se* et ne relève pas de manière absolue d'un modèle ou d'un autre. Un mur peut être vu comme un support pour des affiches, comme construction statique pour porter les étages supérieurs, comme résistance thermique pour garder la chaleur à l'intérieur de la maison, ... Chaque être humain attache sa propre idée à l'objet du monde réel. La communication entre les hommes ne peut se faire qu'à travers des parties communes et partagées d'un concept. Un des objectifs de la modélisation est alors la description formelle de ce concept (généralement exprimé sous forme mathématique).

La deuxième phase est l'exploitation de cette modélisation, qui consiste dans notre cas à « faire de la simulation ». La difficulté de cette phase est l'identification d'un objet réel comme une instance d'un modèle existant. On associe l'idée générale à l'objet concret. Cette reconnaissance de l'objet réel se fait en vue d'une future simulation du système. On est donc déjà guidé par les objectifs de son étude et on applique des simplifications. Le mur NORD de notre bâtiment comme objet concret se voit associé le phénomène conduction thermique monodimensionnelle. La simplification ici est de ne pas considérer les effets acoustiques, par exemple. Pour un système concret, on utilise les modèles construits dans la phase de modélisation, et on les applique à son problème. C'est ici, dans la deuxième phase, que l'on utilise une simulation du système modélisé pour obtenir des résultats qui sont autrement trop difficilement accessibles.

Remarquons que dans le langage courant, la modélisation d'un système comprend la spécification de paramètres pour les différents modules utilisés. Nous considérons cette affectation de valeurs comme une étape de la simulation.

1. Une autre manière de construire un modèle est purement abstraite. Il résulte d'un raisonnement et d'une construction logique sur des modèles et axiomes existants. C'est le cas dés mathématiques et de la philosophie

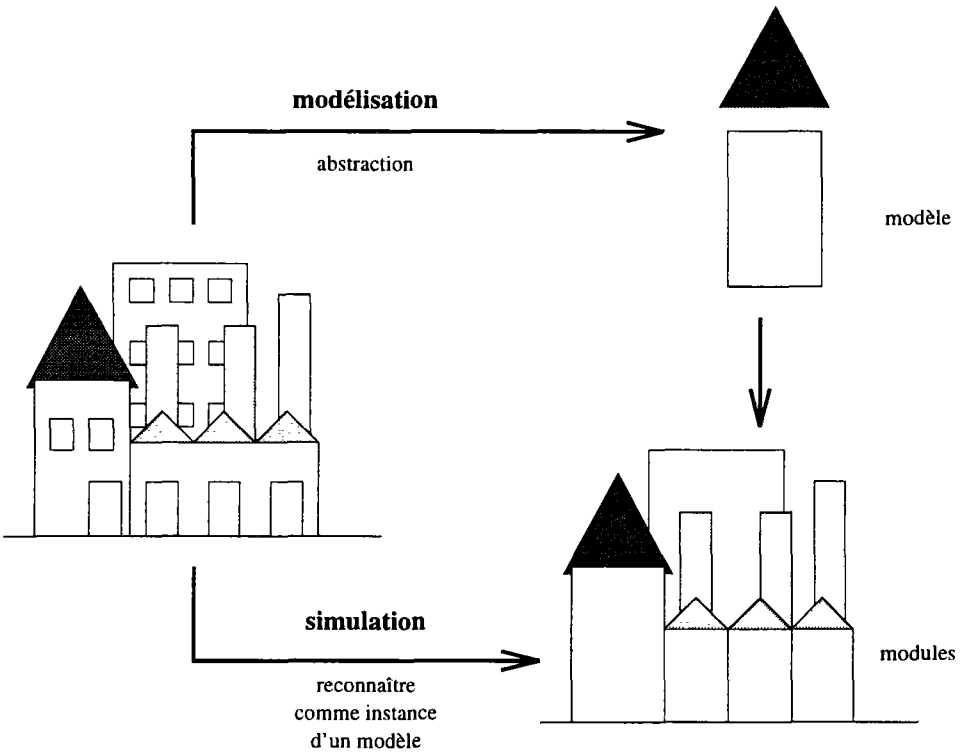


FIG. 2.1 - *Les deux deux phases principales d'une étude de système : la* modélisation *et la* simulation.

Nous associons aussi à la distinction entre ces deux phases une distinction de terminologie : pendant la modélisation – comme le mot l'indique – nous créons des *modèles* de la réalité ; dans la deuxième phase – l'exploitation de modèles, et plus particulièrement la simulation – nous utilisons des *modules* pour décrire un système. Un module est une instance d'un modèle général avec des paramètres concrets. Cette distinction entre les phases et le choix du vocabulaire en résultant seront approfondis dans le chapitre 4.1.

Nous allons donc présenter d'abord des méthodes de modélisation, et surtout les problèmes liés à la complexité. Ensuite les démarches de la deuxième phase d'une étude de système seront regardées plus en détail.

2.1.2 Complexité

La notion de complexité est très liée à celle de système. Par *système* nous entendons un ensemble d'objets qui interagissent, ce qui ajoute une nouvelle qualité à l'ensemble. Il faut insister ici sur l'activité d'un objet. Pour nous un système devient de plus en plus *complexe*, si les interactions entre les objets sont tellement multiplement entrelacées, que les rapports entre les objets et le comportement du système – aussi bien au niveau global qu'au niveau local – deviennent de moins en moins apparents. La difficulté de la complexité est en nous même, c'est un problème d'intelligibilité. Mais la complexité ne s'arrête pas à l'énumération des composants et de leurs rapports dans un système.

La qualité supplémentaire ajoutée à l'ensemble, qui est l'organisation, est une condition de complexité. On peut lire chez E. MORIN, que c'est « *la capacité d'un système à, à la fois, produire et se produire, relier et se relier, maintenir et se maintenir, transformer et se transformer* » [60]. J.-L. LE MOIGNE décrit le *système général* comme : « - *quelque chose (n'importe quoi, présumé identifiable),* - *qui* dans *quelque chose (environnement),* - pour *quelque chose (finalité ou projet),* - fait *quelque chose (activité = fonctionnement),* - par *quelque chose (structure = forme stable),* - *[et] qui* se transforme *dans le temps (évolution)* » [53]. On remarque dans les deux définitions l'accent mis sur l'activité et l'évolution d'un système qui le rendent *complexe*. LE MOIGNE utilise le terme *processeur* pour décrire les composants d'un système afin d'en souligner l'aspect actif. Par ailleurs, ces définitions ne définissent même pas un *système complexe*, mais plutôt un *système* tout court. Le caractère complexe est toujours sous-entendu dans la notion de système. Néanmoins les système en équilibre peuvent être complexes eux-aussi, car leur état et leurs dépendances intérieures sont quelques fois encore plus difficiles à déterminer.

L'*approche systémique* est donc une tentative de développer des théories et méthodes avec lesquelles on peut comprendre, examiner, interpréter, évaluer, construire, manipuler et changer des systèmes, mettre en évidence les dépendances. L'approche systémique cherche à mieux comprendre la complexité organisée en fournissant une vision macroscopique du comportement global, tout en acceptant que certains détails ne sont pas traités en profondeur.

Notre objectif ici n'est pas une approche générale pour tout système. Nous nous intéressons plutôt à la conception et au contrôle des systèmes techniques et artificiels. Par rapport à la sociologie par exemple, l'application de l'approche systémique dans les sciences physiques et naturelles est souvent allégée par des mesures plus facilement quantifiables. Il faut néanmoins remarquer qu'il n'y a pas de système complètement technique. L'interaction humaine existe partout. La complexité des systèmes technique se révèle lors d'une conception plus détaillée, d'une conception sur mesure (nouveaux matériaux et traitements), avec une multiplication de dépendances. Au niveau calculatoire, on travaille souvent dans un champ de fonctionnement restreint, avec des phénomènes dont la description mathématique pose des problèmes à la résolution (phénomènes non-linéaires et non-monotones). Les systèmes techniques et artificiels sont le plus souvent équipés de dispositifs de contrôle de plus en plus sophistiqués, ce qui ajoute encore à la difficulté de les comprendre et de les modéliser.

2.1.3 Modélisation

La complexité croissante des systèmes a augmenté la difficulté de l'analyse et a mis en évidence la nécessité des nouvelles méthodes pour mieux aborder ce type de problème. Aux différents niveaux, on a pu constater des développements, comme les algorithmes sur les grandes matrices en algèbre linéaire, ou le calcul parallèle. Pour la compréhension et la conception des systèmes complexes, l'approche systémique est souvent utilisée.

Née de la cybernétique dans les années 50, l'approche systémique est aujourd'hui appliquée dans des domaines très différents comme la sociologie [20], l'économie [72], [43], l'écologie [79]; de plus depuis quelque temps elle a été (re-)découverte par les ingénieurs [67], [48], [42].

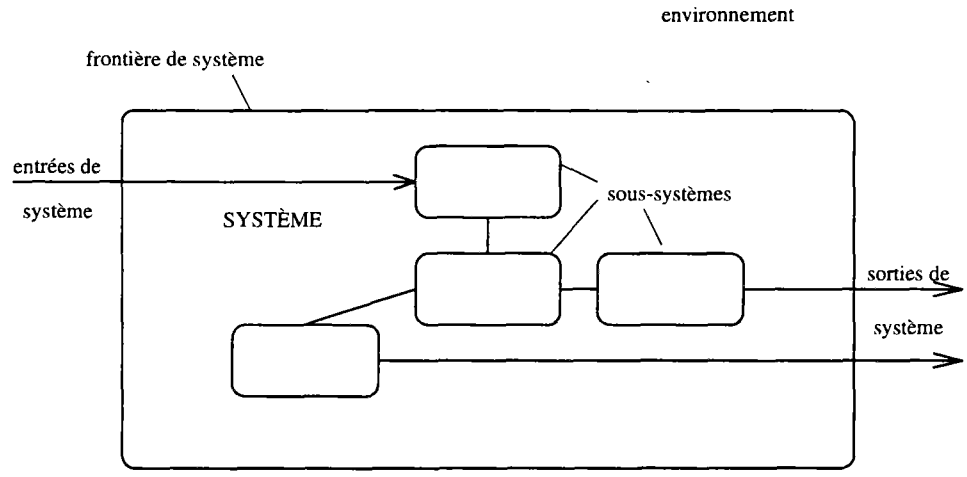


FIG. 2.2 - *Le concept général d'un système avec le découpage de l'environnement, des entrées/sorties et des sous-systèmes. La structure du système, le nombre et le type des éléments et le nombre et le type de raccordements entre les éléments et avec l'extérieur déterminent le système et sa complexité.*

Quelques questions sont typiques pour l'approche systémique, mais elles s'appliquent également à l'analyse d'un système en général. Il convient d'en être conscient pendant toutes les phases d'une étude. Les problèmes suivant nous accompagnent tout au long d'une étude systémique:

Découpage Un système n'est pas isolé de son environnement. Il faut que la partie choisie de la réalité soit plus ou moins autonome (« *système quasi isolé* »). Un critère possible pour le découpage est la minimisation des débits à travers les frontières. Il faut faire attention au fait que les différents types de débits (d'énergie, de matériaux, d'informations) ont en général des frontières à débit minimale différentes. Selon le critère choisi on découpe le système de son environnement en des endroits différents.

Intégralité Cette préoccupation est complémentaire au problème de découpage. N'a-t-on rien oublié de ce qui est important pour la compréhension du système? Une question à laquelle on ne peut pas répondre *a priori*.

Granularité C'est la question du niveau de détail avec lequel nous devons décrire le système. Une granularité trop fine rend le modèle trop lourd et compliqué; une granularité trop grossière ne prend pas en compte des détails importants. Il peut éventuellement s'avérer nécessaire de travailler avec une gamme de modèles à granularités différentes et changer de modèle en fonction des besoins réels.

Échelle de temps Ici il s'agit de choisir des constantes de temps qui sont importantes pour notre problème. (Regarde-t-on l'évolution en millisecondes ou en heures?) Granularité et échelle de temps sont souvent liées. Plus la granularité est fine, plus l'échelle de temps doit être courte. Généralement on peut souvent supposer une inertie plus petite pour un petit système. La conclusion réciproque n'est pas toujours vraie ; même une description grossière du système peut nécessiter une échelle de temps courte.

Représentation La structure mathématique de la description du système définit la représentation. Le choix d'une représentation fait donc partie de la traduction d'une image « intuitive » vers une description mathématique et informatique (nous allons approfondir l'idée d'une représentation à niveaux d'abstractions différents dans le § 4.1). Généralement on est conscient des problèmes mentionnés jusqu'ici. Mais le type de questions que l'on peut se poser au sujet d'un modèle dépend fortement de la représentation choisie.

Prenons comme exemple un mur. On peut modéliser le champ de température comme un problème mono-, bi-, ou tridimensionnel. De plus il faut se décider, entre autre, entre une représentation analytique (au moins pour le cas monodimensionnel), une approche variationnelle, une représentation purement paramétrique ou une discrétisation spatiale. Une représentation par des modes propres est difficile à interpréter physiquement, mais elle apporte des avantages comme la possibilité d'une analyse particulièrement pertinente et riche et d'une réduction de modèle pour la simulation. Une modélisation sans capacité thermique caractérise directement le régime permanent. On peut aussi envisager un modèle qui ne prend pas en compte le comportement thermique mais plutôt ses propriétés mécanique ou acoustiques, bien que notre préoccupation principale reste attachée aux problèmes thermiques. Il existe alors un grand nombre de représentations différentes auxquelles on ne peut pas poser les mêmes questions.

Il peut être éventuellement utile de garder plusieurs représentations à la fois. Selon les simplifications que l'on s'autorise et selon les questions auxquelles on veut des réponses, on peut alors choisir parmi une gamme de modèles pour trouver le modèle le plus adapté. Il se pose alors la question du raccord entre des modèles différents.

Complexité Tout modèle de système est *a priori* une simplification ; la réalité est toujours plus riche que le modèle. Un choix n'est justifiable qu'*a posteriori*, si on prouve (par exemple par expérimentation) que le modèle s'est comporté comme la réalité dans le cadre de la précision choisie et pour l'environnement prévu (grandeurs et fréquences d'excitations).

Autrement dit, si l'on veut obtenir des résultats fiables, il faut que la complexité du système simulant soit plus élevée que celle du système à simuler [6]. Ceci étant impossible, il faut se contenter de résultats approximatifs. Dans la pratique, on ne veut pas simuler *tout* le système,

mais seulement certains aspects. Cela permet des modèles moins complexes que la réalité et quand même fiables dans le domaine observé.

Il faut par ailleurs remarquer, qu'un comportement apparemment complexe n'a pas forcement son origine dans des lois complexes. Par exemple, les systèmes chaotiques comme les ensembles de MANDELBROT sont souvent déterminés par des lois très simples.

Différentes méthodes ont été développées pour appliquer de l'approche systémique à un contexte concret ; par exemple la *méthode de sensibilité* de VESTER mise au point pour l'UNESCO [78]. Cette méthode part de la création du « bon» ensemble de variables, pour déterminer ensuite le réseau d' « interdépendances» entre les modules. Une distinction entre éléments *actifs, critiques, neutres* et *passifs* permet d'évaluer la dynamique du système. Malgré de grands efforts, l'application de l'approche systémique au domaines techniques (génie d'ingénieur) n'est pas encore aussi bien formalisée.

2.1.4 Exploitation de la modélisation : la simulation

Les questions détaillées plus haut sont plutôt liées à des démarches de modélisation qui prennent en compte une future exploitation par simulation. Pour nous, la simulation est la deuxième phase principale d'une étude de système. Cette exploitation de la modélisation est composée de plusieurs étapes, qui ne s'enchaînent pas toujours séquentiellement. Assez souvent on est contraint de revenir à une étape précédente pour réévaluer les décisions.

Un ordinateur peut alléger cette étude de différentes façons. Il intervient surtout dans la phase numérique (phase de la résolution du système d'équations), mais aussi lors de la modélisation des nouveaux composants et de l'interprétation des résultats.

Ces différentes phases sont soutenues sur l'ordinateur par des modules de logiciel différents. Le plus souvent on ne parle pas d'un programme de simulation mais plutôt d'environnement de simulation. Les produits disponibles (et ceux qui sont en cours de développement) ne sont souvent pas limités à un seul programme mais ils sont eux mêmes composés de plusieurs modules, chacun conçu pour traiter un aspect particulier du processus de modélisation. Cet ensemble constitue un environnement de simulation.

On peut par exemple énumérer les phases de la simulation – dans le sens large d'une deuxième phase d'une analyse systémique –, et étudier comment on peut profiter de l'aide d'un ordinateur à chaque étape.

Description générale du problème On commence par une description du problème d'une façon informelle, dans un langage naturel. On observe le système tant que possible; on interroge les personnes concernées (un système n'est jamais purement technique). L'avis des experts et l'extraction de l'information contenue dans la documentation existante relative au domaine comme les article et les livres, approfondit la perception.

Pour mieux structurer cette phase initiale, l'aide de l'ordinateur peut intervenir non seulement au niveau de l'organisation comme par des logiciels de bureautique ou par des logiciels d'organisation des projets, mais aussi par des logiciels de calcul formel et des bases de données électroniques comme des bibliothèques de fonctions.

Identification des processeurs et leurs rapports Puis on cherche à identifier les modules qui constituent le système. Les liens et dépendances (topographiques, logiques, énergétiques) entre les objets sont aussi importants que les objets mêmes. Différentes méthodes existent pour les mettre en évidence (par exemple [78])

L'identification des composants passe par un découpage du système en sous-systèmes. Un découpage peut se présenter de façon très naturelle. On fait par exemple la distinction entre le bâtiment (murs, plancher etc.) et les équipements de bâtiments (par exemple le système de chauffage). Mais on peut aussi essayer de découper le système hiérarchiquement (par exemple les étages, pièces, ...). On retrouve alors ici la notion de granularité et de structure (jusqu'où faut-il couper? quelle est la structure du système?). Une analyse par découpage nous amène à un « graphe » de dépendances et d'appartenances entre les composants.

Un environnement graphique et interactif comme support sur ordinateur améliore certainement la convivialité du travail de la description. On peut imaginer l'intégration d'une méthode schématisée dans cet environnement qui dirige le progrès de la description. Un guide automatisé, géré par un système expert, peut éventuellement simplifier la mise au point d'une description pertinente du système pour l'utilisateur.

Modélisation des phénomènes Pour chacun des composants, un modèle de représentation est nécessaire. On associe aux modules des phénomènes physiques ou des lois empiriques (voir § 2.2.2 pour des différents types de modèle). Les modèles sont une représentation mathématique ou algorithmique qui relient les paramètres et les variables entre elles permettant d'effectuer ultérieurement une étude qualitative.

Ces modèles de calcul avec les équations constitutives identifiées peuvent être stockées dans une bibliothèque de modèles. Une telle bibliothèque contenant des modèles couramment utilisés est pratique, parce qu'elle aide à éliminer les duplications. Cela apporte un gain de temps et nous épargne des redéveloppement de modèles généraux. Néanmoins, les paramètres libres restent toujours à déterminer. En utilisant une telle bibliothèque, la description d'un système consiste finalement à raccorder des modules pré-définis.

Un environnement de simulation doit permettre le développement de nouveaux modèles. Il ne peut pas être uniquement basé sur une « modélothèque » de composants existants. Il doit fournir les outils pour la modélisation comme la description de nouvelles équations, l'exploitation de résultats expérimentaux ou la description algorithmique d'états.

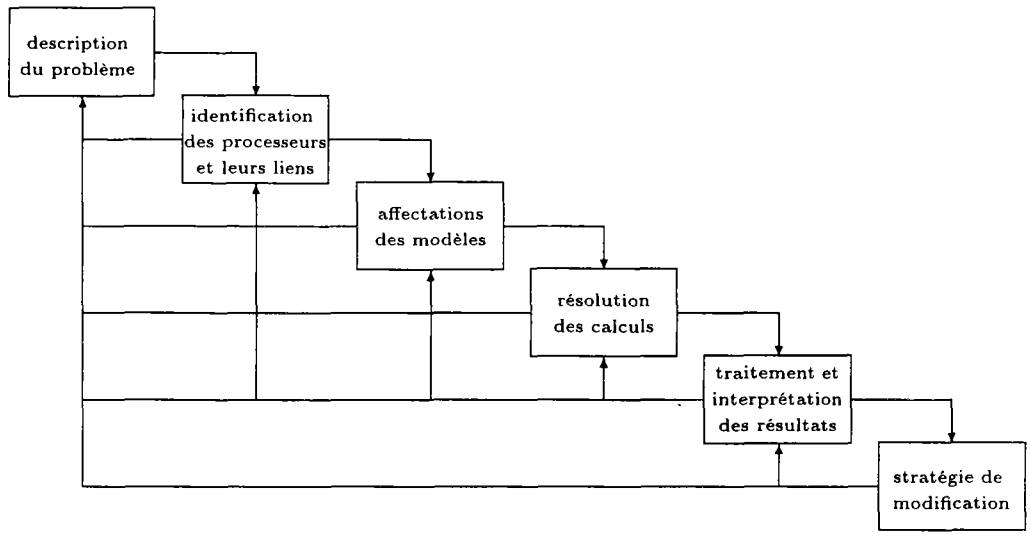


FIG. 2.3 - *Les phases d'une étude de système sont des démarches itératives. Des réflexions sont nécessaires à chaque étape s'il faut revenir à une étape antérieure ou si l'on peut progresser.*

Résolution des calculs La représentation finale d'un problème est un ensemble d'algorithmes sous forme d'un programme d'ordinateur. Pour obtenir des résultats quantitatifs, plusieurs approches existent. Par exemple l'analyse et la synthèse modale permettent de prévoir le comportement dynamique d'un bâtiment rien que par observation de ses modes (voir [52, chap. 1.4]). Dans notre cas, on s'intéresse surtout à la simulation quantitative du comportement temporel du système. En général, il n'y a pas de solution analytique et l'on a recours à des méthodes numérique pour la résolution. Dans le cas de système dynamique, c'est une intégration temporelle des équations différentielles. Aujourd'hui on se repose presque toujours sur la puissance d'un ordinateur pour exécuter les nombreux calculs.

Traitement des résultats Finalement, on peut exploiter les résultats obtenus, par exemple visualiser des champs de température, déterminer les besoins de chauffage d'un bâtiment ou évaluer le risque de surchauffe d'un moteur.

Les résultats se laissent mieux comprendre lorsqu'ils sont représentés par des courbes et des dessins en couleur plutôt que par des listes de nombres. Une présentation graphique de résultats permet de mieux comprendre l'évolution. Ensuite, on peut plus facilement en déduire les conséquences pour la configuration du système, au moins de façon qualitative.

Interprétation des résultats L'interprétation détermine le rôle des éléments dans le système et les propriétés de la structure globale. On peut évaluer la stabilité, la souplesse et/ou l'efficacité du système.

Un support possible par ordinateur peut être un système expert. Mais le degré d'amélioration que l'on peut apporter au système étudié par une

interprétation des résultats par un système expert reste une question ouverte. Il existe quelques recherches dans ce domaine et les epérences sont grandes [55]. Les résultats applicables n'existent pas pour l'instant.

Stratégie Avec cette interprétation, on peut finalement développer une stratégie pour modifier le système. En fonction des sensibilités obtenues et connaissant les réponses probables du système, on a les moyens de modifier ou d'interagir avec le système. Une nouvelle conception du système est possible. Ceci peut être une adaptation des paramètres pour faire évoluer le système, mais on peut aussi penser à une réorganisation du modèle pour adapter la structure du système.

Ces phases de l'exploitation ne se suivent pas forcement dans l'ordre que nous avons mentionné ici. C'est plutôt un processus itératif ou à chaque étape les résultats sont à réévaluer. Si les résultats ne sont pas satisfaisants (par rapport aux valeurs d'expérience ou d'expérimentation) on recommence éventuellement à une phase précédente comme par exemple à l'identification des modules, ou on cherche une meilleure représentation et modélisation des modules existants.

L'outil idéal d'analyse systémique est un environnement qui offre un support spécifique pour chacun des points mentionnés dans les paragraphes précédents. Comme on ne peut pas prévoir tous les problèmes, les outils doivent s'adapter facilement aux besoins des études menées.

La résolution numérique du problème après sa modélisation est une phase parmi d'autres dans l'étude d'un système, comme on l'a vu ici. Assez souvent il s'agit de déterminer le comportement du modèle dans le temps, afin de mieux comprendre le système réel. Dans le paragraphe suivant nous examinons ce processus plus en détail.

2.2 Simulation

Le domaine de la simulation par ordinateur subit en ce moment un phénomène explosif d'innovation et d'élargissement des domaines d'application. Les capacités nouvelles du matériel (« transputers » par exemple) et des logiciels (parallélisme extrême et méthodes d'Intelligence Artificielle) accélèrent encore ce processus. Par ailleurs le prix du matériel ne cesse de baisser et les puissances de calcul accessibles ne cessent de croître.

2.2.1 Objectifs et méthodes de la simulation

Apparemment la simulation est intéressante surtout pour des problèmes complexes, qui sont difficilement abordables par l'expérimentation comme par exemple l'étude du confort des occupants d'un bâtiment. Dans ce cas, la simulation remplace l'expérimentation. Par ailleurs, la simulation est également souvent utilisée pour reproduire des tests, et de cette façon elle peut contribuer

à leur compréhension. Elle peut étendre le champs de ces tests sur le système à étudier et elle permet d'en modifier les paramètres caractéristiques pour comparer plusieurs cas similaires. De la même manière, la structure de l'ensemble et les modèles élémentaires peuvent être modifiés. Comprendre des phénomènes par modélisation peut être un moyen d'acquérir de nouvelles connaissances sur ce domaine. C'est la tâche du modélisateur.

L'exploitation des modèles qui sont validés par simulation ajoute une dimension supplémentaire aux tests. Cela peut donner des indications sur les actions futures à mener et donner des possibilités d'extrapolation. De cette façon la simulation est une aide précieuse à la conception des nouveaux systèmes, et incombe donc à l'utilisateur du logiciel de simulation.

Les objectifs des études qui reposent sur la simulation sont :

- la compréhension de la structure et des rapports à l'intérieur d'un système.
- l'analyse et la prédiction du comportement du système.
- la conception et le dimensionnement de l'équipement.
- l'aide à l'expérimentation, qui peut éventuellement finir par ...
- le remplacement de l'expérimentation, souvent très coûteuse et même quelque fois impossible.
- la validation des systèmes de contrôle.

Les résultats de simulation sont évalués en fonction de différents critères qui sont souvent concurrents. Pour une étude thermique de bâtiment cela peut être la qualité, le confort, les coûts ou la consommation.

Pour analyser la réalité par l'expérimentation et interpréter les résultats, on a développé des modèles qui sont des simplifications de cette réalité. Dans les sciences physiques, les modèles sont surtout des modèles mathématiques. Mais il existe aussi d'autres formes de représentation, comme un champ de distribution de probabilité, des règles pour un système expert, ou des connaissances stockées dans un réseau neuronal. En utilisant ces modèles, on peut calculer le comportement dans le temps du système. Par rapport à l'expérimentation, la simulation est plus simple, plus pratique et surtout moins chère. Dans le domaine du bâtiment, elle est souvent la seule possibilité d'étudier un système en raison des grandes dimensions spatiales qui induisent la taille du problème. Elle aide à la conception de l'équipement et de l'enveloppe. Simultanément elle apporte des connaissances sur la structure et sur les dépendances entre éléments à l'intérieur du système.

2.2.2 Types de simulation

La simulation est la reproduction artificielle – informatique essentiellement – d'un comportement réel. On suppose que cette image de la réalité se comporte

« de la même manière » que le phénomène à étudier. C'est le cas pour les simulations basées sur les modèles physiques.

Simulation analogique – simulation digitale La simulation peut être une expérimentation à échelle réduite qui permet l'extrapolation à l'échelle réelle. On peut observer cette approche dans les tunnels de vent par exemple. Par ailleurs on fait des expérimentations sur des systèmes plus simples, mais dans lesquels il existe une analogie des lois physiques. Les ordinateur analogues en sont un exemple. Parfois cette approche est appelée une *simulation analogique*. Aujourd'hui on a le plus souvent recours à la puissance de calcul des ordinateurs, la *simulation digitale*. L'ordinateur est un outil qui permet un large éventail d'approches très différentes pour mener des calculs.

Temps discret – événements discrets Les systèmes thermiques sont en général des systèmes dynamiques, une simulation du régime permanent est insuffisante. La dynamique des phénomènes thermiques au sein d'un système est souvent continue. Deux types d'approches existent pour prendre en compte cette dynamique : simulation par *temps discret* et simulation par *événements discrets*.

Dans le premier cas la simulation passe par une discrétisation du temps. L'horloge des modèles « *avance en segments discrets en passant d'une valeur à la prochaine avec un pas de temps spécifié* » [81]. Dans ce cas, la simulation consiste à calculer l'état du système à chaque pas de temps. Un changement d'état du système est déterminé aux intervalles définis.

La simulation par *événements discrets* utilise un temps continu. La simulation n'est pas avancée par des pas de temps, mais plutôt par une liste d'événements à venir. Cette liste « *contient des instants d'horloge auxquels on assigne aux composants un nouvel état déterminé d'une manière interne* » [81]. L'avantage d'une simulation par événements discrets est une souplesse qui évite la perte de précision d'un pas de temps trop long et la lourdeur d'un pas de temps trop fin. Les inconvénients d'une simulation par temps discret sont surmontés aujourd'hui par des pas de temps variables, gérés pour diminuer l'erreur de calcul. La simulation par événement discret est basée sur deux conditions qui ne sont que rarement vraies pour des systèmes thermiques continus : 1) l'apparition d'un événement peut être déduite à partir de l'apparition d'autres événements. 2) Si l'apparition d'un événement ne peut pas être prévue, les composants ne changent pas d'état, sauf si ce changement d'état est déclenché par le changement d'état d'un composant qui a été programmé [81]. Dans ces conditions, une simulation a lieu avec des pas de temps irréguliers et l'horloge est avancée irrégulièrement à l'instant du prochain événement. Habituellement, on installe un gestionnaire d'événements qui envoie des signaux aux modules concernés.

Dans presque toutes les applications modernes on trouve des dispositifs de régulation qui, quant à eux, se laissent très bien décrire par une représentation

aux états discrets. Une simulation générale de systèmes thermiques devrait permettre un mode mixte. Dans le cadre du présent travail nous nous intéressons à une simulation déterministe, continue et dynamique, par temps discrétisé.

2.2.3 Types de modèle et algorithmes de simulation

Il existe différentes façons d'effectuer une simulation. La méthode à choisir pour la simulation dépend fortement des types de modèles qui sont utilisés pour décrire le système, ses composants et ses liens intérieurs.

- Un *modèle de connaissance* est une construction formelle déduite d'une théorie. Le plus souvent, et en particulier pour ce qui concerne la thermique, cette construction utilise une formulation mathématique. Le langage mathématique est un des moyens pour une description abstraite de la réalité.

  La modélisation des problèmes continus dans l'espace, dans le temps et/ou d'autres dimensions se fait généralement à l'aide d'équations différentielles partielles. On parle des champs qui sont décrits par des *paramètres distribués*. Un flux d'énergie ou de matière peut apparaître dans toute direction.

  Si l'espace est discrétisé ou si le problème ne dépend pas des paramètres continus dans l'espace, on peut représenter le système par des équations différentielles ordinaires. Les paramètres sont concentrés localement (*lumped parameters*). C'est le cas des système tels que l'équipement de chauffage ou de climatisation dans un bâtiment. Ici, le flux d'énergie ou de matière est restreint à certaines voies discrètes.

  Si à la fois l'espace et le temps sont discrets (ou discrétisés), on obtient des équations algébriques orientées par bloc. C'est le seul type d'équation qui puisse être résolu sur un ordinateur. Tout autre type d'équation doit être transformé en équation algébrique pour pouvoir le résoudre sur ordinateur. Ce type d'équation est surtout obtenu par une discrétisation de l'espace. Différentes techniques de discrétisation spatiale existent comme les différences finies, les formulations variationnelles, les éléments finis, ou les volumes finis.

  Associés à ces types de modèles (paramètres distribués, paramètres concentrés, paramètres par bloc) sont des composants typiques de système. Un milieu continu est décrit par un système d'équations différentielles partielles ou les paramètres sont les propriétés locales du problème. Cela est le cas, par exemple, de l'équation différentielle instationnaire pour la conduction, construite à partir du bilan thermodynamique d'un élément différentiel :

  $$\rho A(x) c_p \frac{\partial T}{\partial t} = \frac{\partial}{\partial x}(\lambda A(x) \frac{\partial T}{\partial x})$$

  La conductivité λ, la densité ρ et la surface A sont des paramètres continus qui peuvent changer de valeur en fonction de la variable libre x.

Le comportement des éléments d'un circuit (électrique, thermique, hydraulique, ...) est bien représenté par des équations différentielles ordinaires ou même algébriques. Ses paramètres sont des attributs, des grandeurs spécifiques de l'élément.

- Une autre catégorie de modèle est l'ensemble des *modèles empiriques.* Ces modèles s'expriment de la même façon que les précédents, mais ils sont construits différemment, en simplifiant les équations physiques et en ajustant les paramètres empiriquement. Pour la modélisation de la convection naturelle simplifiée, on peut exprimer le débit traversant la facette séparant deux cubes par :

$$\dot{m} = Sk\rho\Delta p^n$$

où k et n prennent des valeurs empiriques pour décrire la perméabilité de la facette et le type de l'écoulement.

- Les modèles *paramètriques* sont obtenus en construisant des équations qui approchent le mieux possible l'expérimentation. Les paramètres sont ensuite calibrés par identification. Ce type de modèle n'aide pas à expliquer les phénomènes physiques, car les modèles sont créés d'une manière inductive par rapport aux modèles de connaissance dont la construction est déductive.

Nous allons maintenant présenter une évaluation des environnements de simulation existants et estimer leur capacités à couvrir les différentes phases d'une simulation.

2.3 Quelques méthodes et environnements de simulation actuels

Pour la simulation numérique, les équations constitutives des modèles sont traduites en un programme d'ordinateur grâce à des algorithmes numériques. Ces dernières deux décennies ont vu apparaître de nombreux programmes de ce type ; quelques uns sont présentés dans les paragraphes suivants.

Nous voulons distinguer deux types différents d'environnements de simulation. Des environnements avec un traitement direct et des environnements avec une génération intermédiaire de code.

- Le premier groupe contient les environnements globaux ; ils travaillent directement sur une entrée qui en général contient la description du problème. Les calculs de simulation sont immédiatement effectués par le même programme. Nous appelons ces environnements des traitements « à une étape ».

- Il existe ensuite les environnements qui procèdent en plusieurs étapes. Ils traitent d'abord la description du problème. À l'aide de cette description, ils génèrent ensuite un programme de simulation approprié. C'est ce programme généré qui exécute effectivement la simulation. Nous les qualifions de « multi-étape ».

Par rapport à la première approche, la deuxième voie nécessite une étape supplémentaire de compilation. Mais généralement elle génère des simulateurs plus efficaces grâce au choix de méthodes numériques mieux adaptées au problème particulier. Ils essaient de remplacer le travail de l'analyste numéricien. Souvent, les environnements multi-étapes sont plus modulaires et mieux adaptés aux phases de la simulation.

Nous avons testé les logiciels de simulation TRNSYS, TUTSIM, Neptunix, Spark et Zoom. Dans les paragraphes suivants, nous allons les présenter brièvement. Une évaluation de leurs avantages et inconvénients sera donnée pour ce qui concerne la description du système, la possibilité d'étendre et alimenter la bibliothèque de modèles, et finalement la pertinence des résultats obtenus. TRNSYS, TUTSIM et Zoom appartiennent au premier groupe des simulateurs à une étape. le deuxième groupe des environnements avec génération du code est composé de Neptunix et Spark.

2.3.1 TRNSYS

L'environnement de simulation le plus connu et utilisé au niveau international dans le domaine du bâtiment est probablement TRNSYS [73]. Le programme TRNSYS fut développé par le Solar Energy Laboratory de l'Université du Wisconsin, USA en 1975. Fin 1990, la version 13.1 était disponible.

TRNSYS définit un système comme un ensemble de composants connectés entre eux. À l'origine il était conçu pour la simulation des systèmes d'équipement de bâtiment (HVAC), des systèmes solaires. Il est aujourd'hui disponible dans une version généralisée pour la simulation d'un bâtiment complet.

Le programme principal contient une collection de 75 composants pré-définis (par exemple capteurs solaires, radiateurs, bâtiments multizones, etc.) dont la représentation interne est documentée sous forme écrite, dans un classeur. Dans le fichier d'entrée (appelé TRNSYS desk) l'utilisateur indique les composants qu'il veut utiliser pour simuler son système. Pour chaque composant, il spécifie les paramètres (par exemple la valeur d'une capacité calorifique) et les connexions entre les composants. Ces connexions déterminent l'ordre de calcul. Pour stocker les résultats, il y a la possibilité de connecter la sortie d'un composant avec une «imprimante» virtuelle qui place dans un fichier l'évolution de cette variable pendant la simulation. Il faut que chaque sortie d'un composant soit connectée avec une entrée d'un autre composant. Il y a des composant qui lisent un fichier et qui alimentent le système en données externes (par exemple les données météorologiques).

Les modèles associés aux composants et disponibles dans la bibliothèque sont désignés sous le nom de types. Il y a un sous-programme pour chaque type avec une structure fixe de paramètres. Ce sous-programme est appelé pour chacun des composants de ce type à chaque pas de temps, et éventuellement à chaque pas d'itération (pour les composants contenant des équation différentielles).

Extension du système

Si le choix des composants pré-définis ne suffit pas pour un problème donné, l'utilisateur peut écrire ses propres modules de comportement sous la forme d'un sous-programme FORTRAN. Puisque l'interface entre les composants et le programme principal est bien spécifié [73] et les calculs d'intégration sont presque indépendants des calculs dans les sous-programmes, on peut les programmer sans trop de problèmes numériques. Cependant l'utilisateur doit savoir programmer en FORTRAN et connaître la structure des données interne de TRNSYS.

Méthode numérique

Dans la boucle principale où TRNSYS appelle tous les composants pour un pas de temps, TRNSYS peut aussi intégrer une équation différentielle dans les sous-programmes. La méthode d'*Euler modifiée*[2] est utilisée pour calculer un nouveau vecteur d'entrées en fonction des valeurs des dérivées par rapport au temps.

La routine qui interprète le fichier d'entrée classe les composants dans un certain ordre. Au début sont placés les composants qui ont des variables dépendantes du temps et nécessitent l'intégration numérique. Puis suivent les composants qui n'ont pas de dépendance du temps (dont les relations sont résolues par une méthode de *Newton-Raphson*[3]) et finalement les composants qui écrivent les résultats dans un fichier (output units). Ceci est en fait un algorithme de résolution des systèmes d'équations différentielles ordinaires avec équations algébriques.

TRNSYS effectue l'intégration numérique seulement pour les composants qui sont marqués parmi les fonction dépendantes du temps dans la description du fichier d'entrée. Sur l'ensemble des modules, TRNSYS opère une relaxation (avec des options d'accélération ou ralentissement en cas de difficultés numériques).

Évaluation

Un des grands avantages (mais aussi inconvénients) de TRNSYS est son âge. Dans le temps, a été développée toute une panoplie de modèles pour des usages

2. voir annexe D.2 pour une description des méthodes de résolution numérique des équations différentielles ordinaires.

3. les méthodes de résolution de systèmes d'équations non-linéaires se trouve dans l'annexe D.1.

très différents. Un club d'utilisateurs assiste le débutant et lui donne accès à une grande bibliothèque de modèles non-standard, ainsi qu'aux dernières modifications. Le programme, écrit en FORTRAN66, est très portable et « tourne » sur presque toutes sortes d'ordinateurs. L'utilisateur type est le thermicien qui ne veut pas développer ses propres modèles, mais plutôt utiliser ce qui existe. En général, il n'est pas nécessaire pour l'utilisateur de s'occuper des équations internes ou des problèmes numériques.

Un point faible de TRNSYS est certainement l'interaction avec l'utilisateur. L'entrée de données est très pénible suite à une écriture désuète. L'utilisateur doit scrupuleusement respecter l'ordre et l'emplacement des chiffres qui décrivent les liaisons et les paramètres. Des développements récents comme CSTBât essayent de faciliter cette lourde tâche. Par ailleurs, les modèles TRNSYS sont conçus d'une manière fixe pour ce qui concerne le nombre et la qualité des paramètres et des entrées/sorties. Si une variable qui existe déjà comme sortie doit devenir une entrée, l'utilisateur doit écrire une nouvelle sub-routine FORTRAN pour effectuer cette nouvelle mission.

2.3.2 Bondgraphs / TUTSIM

La méthode des « bondgraphs » contient une approche *top-down* pour la simulation des systèmes. Son application suppose d'abord une description globale du système, et ensuite elle examine les détails des composants constitutifs.

Le système des « bondgraphs » est plutôt une approche pour la compréhension d'un système technique quelconque qu'un environnement de simulation au sens strict. Les trois principaux domaines d'application sont l'électricité, la mécanique et l'hydraulique. Son usage n'est pas très répandu dans le domaine de la thermique, car on doit alors gérer des flux d'entropie, ce qui n'est pas très naturel.

Les bondgraphs sont une représentation « par réseau » d'un système. Le réseau comporte les éléments constitutifs du système et les liaisons qui les relient. Plus particulièrement les bondgraphs sont une représentation topologique des débits énergétiques entre les éléments discrets d'un système. Pour cette schématisation on utilise un graphisme symbolique avec :

- des signes (lettres) représentant les composants du système,
- des liens orientés entre les composants appelés « bonds », et
- des connexions entre des liens (voir les *triports* plus bas)

Plusieurs composants peuvent être regroupés dans un seul bloc. Le système, une fois schématisé par les bondgraphs, se traduit facilement en un système d'équations différentielles ordinaires et ensuite en un programme de simulation (voir plus bas pour les supports sur ordinateur). La schématisation par graphes constitue une phase initiale avant l'étude quantitative du système. Elle aide à la

construction du système d'équations ; aucune méthode numérique n'est spécifiée par les Bondgraphs.

Par exemple, la schématisation simple d'une voiture contient le moteur, la boîte de vitesse et les roues. Leurs relations peuvent être schématisées par des bondgraphs comme dans la figure 2.4.

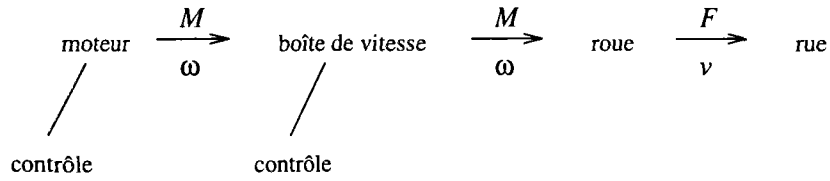


FIG. 2.4 - *Schématisation simple d'une voiture.*

Chaque mot représente ici un composant principal ou un sous-groupe. Les flèches entre les mots sont les « bonds » qui indiquent une « puissance directionnée ». Les traits droits signifient un flux d'information (par exemple le contrôle du moteur par l'accélérateur).

Les bonds contiennent deux variables principales. Elles sont appelées *effort* et *flux*. Leur produit est en général une puissance. Sont toujours associées les variables d'*impulsion* et de *déplacement* qui sont des intégrales d'effort et de flux dans le temps. On dessine un bond comme une flèche qui indique la direction d'écoulement de l'effort et du flux. Dans le domaine de l'électricité on utilise naturellement tension U (= *effort*) × courant I (= *flux*); en mécanique sont utilisés force F (*effort*) × vitesse v (*flux*) ou moment M× vitesse angulaire ω comme dans la fig. 2.4. En thermique les équivalences ne sont pas immédiates : température T (= *effort*) × flux d'entropie $\dot{S}$ (= *flux*). Bien que satisfaisant intellectuellement, le travail avec un flux d'entropie n'est pas naturel pour un thermicien. On peut aussi utiliser la température et le flux de chaleur s'il n'y a pas de conversion d'énergie thermique / mécanique dans le système.

En ce qui concerne les composants, il y a trois classes standards que l'on distingue par le nombre de liaisons possibles : monoport, diport, tri- ou multiport, qui ont respectivement un, deux, trois ou plusieurs points de connexion. Un composant est plus qu'une simple relation mathématique. Il a trois niveaux de description:

- équation constitutive
- relation d'énergie et de puissance
- réversibilité ou non (au sens du 2^{e} principe de la thermodynamique)

Pour les composants monoports on utilise une forme généralisée des composants électriques : R (*résistance*, pour les relations de proportionnalité), C (*capacité*, pour les relations différentielles) et I (*inertie*, pour les relations intégrantes). Des sources d'effort (SE) et des sources de flux (SF) sont également des composants monoports. Chacun de ces composants a exactement une seule

possibilité de raccordement (voir fig. 2.5 pour l'exemple d'un bondgraph représentant un mur bicouche).

À titre d'exemple, regardons les trois niveaux de description pour un élément C: l'équation constitutive relie la charge et la tension (électrique) (soit en général l'*effort* et le *déplacement*) et est indépendante du temps. Les éléments C conservent l'énergie, mais pas la puissance. La puissance est absorbée en chargeant l'élément qui stocke l'énergie. Cela change l'état de l'élément. En revenant à l'ancien état l'énergie est rendue à travers le *bond*. Les éléments C sont réversibles.

Il y a seulement deux composant avec deux liaisons: les transformateurs (TR) et les gyrateurs (GY). Les transformateurs gardent la proportionnalité entre les efforts entrants et sortants (transformateur électrique, boîte de vitesse). Un gyrateur indique une proportionnalité entre l'effort entrant et le flux sortant (ou flux entrant et effort sortant). Le plus souvent il est utilisé pour un transfert de puissance entre deux domaines physique. par exemple un moteur électrique dont la rotation (flux sortant) est proportionnelle à la tension d'entrée (effort entrant).

Les triports sont des jonctions entre plusieurs bonds. Les jonctions parallèles (appelées p ou 0) ont une égalité de l'effort dans toutes les branches et la somme de flux est nulle. Les jonctions en série (appellées s ou 1) qui ont une égalité du flux dans toutes les branches et la somme d'efforts est zéro (voir fig. 2.5). [77, chap. 2.3].

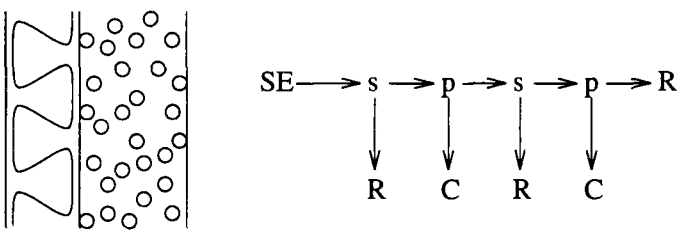


FIG. 2.5 - *Un mur bicouche sollicité par une température à gauche ; chacune des couches représentées par une capacité thermique, séparées par des résistances thermiques.*

Causalité

Un principe fondamental des bondgraphs est la causalité. Par indication de la causalité, on donne un sens aux formules constitutives des composants qui sont *a priori* non-orientées. C'est une phase nécessaire pour passer à la simulation. Un tiret supplémentaire est ajouté, soit au début soit à la fin de chaque bond pour indiquer la « causalité » de l'élément connecté. La causalité indique quelle variable de la liaison est calculée en fonction de l'autre. De cette façon, le sens (l'ordre) des calculs peut être déterminé.

Voyons un exemple pour un élément du type *proportionnalité* dont l'équation constitutive sans indication de la direction de calculs est:

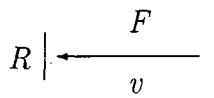
$$\textit{vitesse} = \textit{force} \;/\; \textit{résistance}$$

ou

$$force = résistance \cdot vitesse$$

si on prend un cas d'application en mécanique où le flux est une vitesse et l'effort est une force.

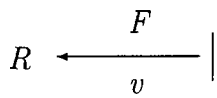
On choisit l'une des deux représentations par le tiret de causalité. Si le tiret est près du composant, le composant calcule le flux en fonction de l'effort (fig. 2.6).



$$v = f(F) = \frac{F}{R}$$

FIG. 2.6 - *Causalité pour calculer le flux en fonction de l'effort*

Si le tiret est de l'autre côté du composant, le composant calcule l'effort en fonction du flux (fig. 2.7).



$$F = f(v) = v \cdot R$$

FIG. 2.7 - *Causalité pour calculer l'effort en fonction du flux*

Quelques causalités sont imposées, sinon le système d'équations n'est pas soluble. C'est évident pour les sources, une source d'effort ne peut pas livrer un flux en fonction de l'effort. Les jonctions déterminent également des causalités. Une jonction parallèle a exactement *un* effort qui entre, les autres sortent ; une jonction en série a *un* flux entrant, sur les autres branches ils sortent. Le choix de la causalité est quelque fois arbitraire. Pour des raisons numériques, on choisit de préférence une causalité intégrante et on évite les causalités qui entraînent des dérivations numériques.

La méthode de bondgraph aide à la description des systèmes. Elle sert à déterminer les interdépendances entre les équations constitutives et pour trouver un bon ordre de calcul, cohérent avec le principe de causalité.

Pour l'étude d'un système et la description de la simulation on dessine d'abord un bondgraph du système à examiner avec les symboles définis (composants et bonds). Il existe ensuite plusieurs logiciels qui permettent de saisir directement une description formelle d'un bondgraph (voir fig. 2.8) et qui exécutent la simulation : TUTSIM (voir en bas) éventuellement avec FANSIM [35], ENPORT [69] et le couple CAMP/ACSL [18].

TUTSIM

Parmi les logiciels d'application de la méthode des bondgraphs, nous avons choisi TUTSIM et sa version pour PC [57] en raison de sa disponibilité et sa facilité d'utilisation.

L'utilisateur de ce programme décrit les éléments de son problème dans un ordre quelconque à l'aide d'un programme de saisie spécial. Non seulement les

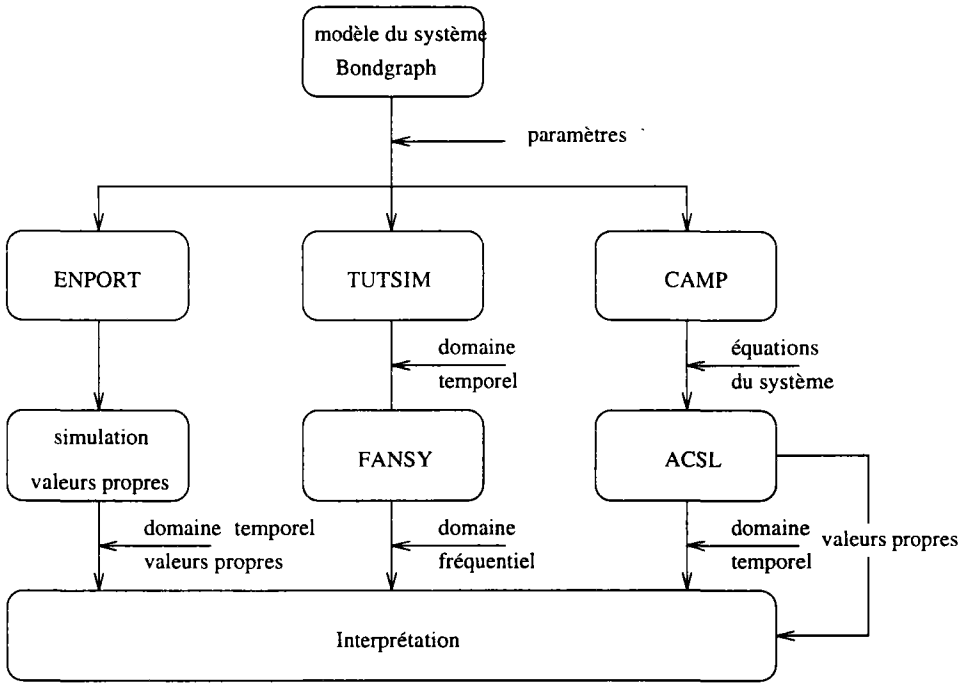


FIG. 2.8 - *Une carte de logiciels pour obtenir des résultats d'un bondgraph.*

éléments de base des bondgraphs sont permis, mais aussi un grand nombre d'autres composants qui sont apparemment des restes du temps des ordinateurs analogiques. Après la saisie du problème, TUTSIM fait des contrôles de calculabilité, et il simplifie le graphe des connexions.

L'exécution effective de la simulation passe par un algorithme d'*Euler* et TUTSIM crée un fichier de résultats pour un postprocesseur. Mais un des avantages de TUTSIM est l'affichage direct des résultats sur un écran graphique. L'utilisateur peut temporairement changer les paramètres des équations. Ceci rend le programme très souple et interactif, car on peut observer et on obtient directement et en temps réel les conséquences d'une modification du système sur les résultats.

Évaluation

La méthode des bondgraphs fournit une description pertinente des systèmes continus par une représentation bien formalisée – des « flux de puissance» entre des composants discrets. L'analyse que l'on est obligée de faire pour dessiner le graphe et décider les causalités permet une meilleure compréhension des liens et dépendances à l'intérieur du système. Cette méthode est un outil de modélisation puissant, c'est à dire du développement d'une représentation mathématique du système. C'est la raison pour laquelle on ne trouve pas de bibliothèque de modèles pré-définis.

Un composant d'un bondgraph est limité à une fonction multiplication, intégration, dérivation, ou une combinaison de celles-ci. Cela permet la représen-

tation de toutes les équations continues. Pour représenter des états discrets et logiques, la méthode est encore mal adaptée. Si une fonction est définie sur différentes plages ou si elle agit en fonction d'un seuil, il faut utiliser des composants non-standards et non-définis dans le concept (mais disponible dans TUTSIM).

Un environnement graphique pour la saisie d'un système n'existe pas encore, bien que le formalisme s'y prête. Par contre, la présentation graphique des résultats et la souplesse dans la modification des paramètres est bien réussie dans le produit TUTSIM.

2.3.3 Neptunix

Neptunix est un environnement de simulation à deux étapes. Il génère un programme solveur pour simuler des systèmes qui sont décrits par un système d'équations algébro-différentielles (SEAD). Son grand avantage est la possibilité de pouvoir traiter des équations non continues par un changement d'état du simulateur.

L'utilisateur doit développer un modèle mathématique de son problème avant l'exploitation de Neptunix, puisque le programme ne fournit aucun support, comme des formules pré-définies par exemple. À partir de la description du modèle mathématique faite dans son propre langage de pilotage [22], Neptunix génère des fichiers source FORTRAN pour construire le programme final de la simulation. On distingue alors deux phase différentes :

- la phase de génération du simulateur à partir de la description mathématique de son comportement,
- la phase de l'exploitation du simulateur ainsi construit.

Les paramètres de la deuxième phase, comme la durée de la simulation et certains autres paramètres des équations, peuvent être contrôlés par un autre langage de pilotage [21].

Puisque Neptunix est plutôt un solveur des SEAD, il n'est pas limité aux problèmes thermiques. Tous les systèmes que l'on peut décrire par des SEAD peuvent être traités par Neptunix.

ALLAN

ALLAN est un interpréteur graphique conçu et développé par Gaz de France [23]. Son objectif est de décharger les ingénieurs d'études de l'analyse numérique et de la programmation informatique liées à l'utilisation des modèles. En fournissant les outils de gestion d'une bibliothèque de composants, de sous-systèmes ou même de systèmes complets, ALLAN est un outil qui facilite la réutilisation des études réalisées précédemment.

ALLAN est essentiellement un préprocesseur pour un langage de simulation. Pour l'instant, des sorties existent pour Neptunix et Astec.

Trois type de composants sont connus par ALLAN :

- des *composants simples*, dont la fonction est décrite par des équations algébriques ou différentielles.
- des *composants composés*. Dans un composant composé sont définis les raccordements entre des sous-composants (simples ou eux-même composés).
- des modèles de simulation. Leur schéma fonctionnel est utilisé pour générer le code de résolution dans le langage Neptunix. Avant cette étape ALLAN contrôle si toutes les initialisations et les paramètres propres à la simulation sont déclarés.

À travers l'interface ALLAN l'utilisateur peut totalement piloter le solveur aussi bien pour la génération du code que pour la simulation et le post-traitement. Des commandes spéciales permettent la visualisation graphique des résultats.

En cas de problèmes numériques ou suite à des disfonctionnements d'ALLAN, ou tout simplement pour plus de souplesse, l'utilisateur peut modifier directement les fichiers générés pour agir sur les équations ou pour guider l'algorithme de résolution.

Évaluation

Le grand avantage de Neptunix est la génération automatique d'un solveur spécialement dédié au problème. Ceci permet d'utiliser des algorithmes optimisés pour le système à traiter. Comme résultat, on obtient un code très efficace et sûr. Un autre point fort est la connexion avec un *automate* pour gérer des discontinuités dans les équations. Ce mode mixte permet la combinaison de fonctions continues, définies par intervalles, et de fonctions discrètes. La définition des équations sous forme implicite évite le problème d'orientation et d'inversion des équations. Suite à la génération de code spécifique, on perd en souplesse si l'on veut modifier le système ou même une seule équation, car un nouveau programme doit être généré et compilé à nouveau, bien que cela se passe automatiquement. Il n'est pas très commode pour les calculs vectoriels et matriciels tels que ceux qui résultent d'une modélisation en différences finies, par exemple.

De même que pour TRNSYS, la définition du système d'équations et la saisie des paramètres sont lourds. Le travail au niveau mathématique ne présente aucun rapport avec le problème physique de départ. Ces inconvénients sont effacés par le préprocesseur ALLAN. Cet outil permet la création d'une bibliothèque de modèles. Une interface graphique conviviale qui existe à la fois pour la saisie de modèles, pour la description d'un système composé et aussi pour l'affichage des résultats, facilite le travail avec Neptunix.

2.3.4 EKS / Spark

Le nom EKS est l'abréviation de « Energy Kernel System », un système général pour la simulation des systèmes énergétiques. À l'intérieur de l'EKS se trouve le Simulation Problem Analysis Research Kernel (Spark)[4] en cours de développement par le Lawrence Berkeley Laboratory et l'University of Fullerton en Californie [16].

Spark est un système de logiciels pour la description et résolution des équations algébro-différentielles non-linéaires. Il est composé de plusieurs logiciels modulaires. L'utilisateur a quatre points d'interaction avec ce système : il peut définir des objets (qui peuvent être élémentaires ou composés et qui sont accessibles dans une bibliothèque), il peut définir un problème à résoudre par assemblage des objets; il doit spécifier des données pour l'exécution (données météorologique, paramètres ...) et il peut spécifier les sorties souhaitées.

Les objets de base sont des équations pour Spark. Chaque équation est un objet élémentaire séparé. À un niveau supérieur existent des *macros* qui rassemblent plusieurs équations ou macros dans un objet. Si une macro fait référence à une sous-macro, on obtient une structure hiérarchisée de description. L'algorithme interne de Spark résout cette hiérarchie en mettant toutes les équation « à plat ».

L'utilisateur pourra décrire les problèmes par connexion d'objets à l'aide d'une interface graphique qui est en cours de développement. Pour l'instant Spark est encore limité à une écriture « à la main » des fichiers nécessaires ; on peut cependant utiliser un préprocesseur (voir plus bas). L'éditeur graphique produit un fichier contenant les spécifications en NSL [3], un langage pour les spécification d'un réseau. Spark détermine un ordre de calcul pour la résolution du système d'équations et en fonction de ces résultats, il génère le programme de simulation adapté au système d'équations. Ce simulateur lit les données et calcule la solution. Un postprocesseur peut lire les résultats et les afficher grâce à une interface graphique.

Algorithmes internes

Spark analyse le fichier de spécifications qui lui est soumis, et traduit de manière interne les équations sous forme de graphe. Une équation est représentée par un nœud, et une variable par un arc. Un arc relie deux nœuds si la variable associée est commune aux deux équations associées.

Ensuite un algorithme de théorie des graphes, dit algorithme de DINIC [4], couple chaque équation à une et une seule de ses inconnues, qui dès lors sera calculée exclusivement par son équation associée. Cela revient à orienter les arcs du graphe, l'arc sortant d'un nœud si il correspond à la variable couplée à l'équation-nœud, et rentrant sinon. Puis un autre algorithme heuristique, dit algorithme de LEVY-LOW [54], détermine un ensemble « de petite taille» d'arcs

4. noyau de recherche de l'analyse du problème de simulation (!)

tels que couper ces arcs élimine tous les cycles du graphe. Cela revient à trouver un « petit » ensemble de variables telles que leur connaissance exacte permet de dériver toutes les autres inconnues de façon explicite. Spark détermine donc un ensemble de variables d'itération du problème, de taille n_{it} inférieure à la taille du problème. Il suffit donc de résoudre le problème non linéaire réduit, avec seulement les variables d'itération comme inconnues. Comme résoudre un problème non-linéaire de taille N passe par l'inversion de la matrice Jacobienne du système (voir plus bas), qui a un coût $O(N^3)$, on réduit le temps calcul d'un facteur $O(r^3)$, ou r est la réduction $\frac{N}{n_{it}}$.

Une fois le problème réduit résolu, la solution est propagée aux inconnues restantes.

Interface symbolique

Si on dispose d'un logiciel de calcul formel tel que Macsyma [59] ou Maple [19], on peut l'utiliser comme préprocesseur pour Spark. L'interface symbolique crée du code qui ensuite peut être utilisé par Spark. La puissance des outils de calcul formel permet d'utiliser directement des représentations non-orientées des objets de base que sont les équations.

Il existe en effet des commandes pour créer toutes les formes explicites d'une équation implicite. L'utilisateur peut spécifier de « mauvaises inverses », c'est à dire une liste des variables de l'équation en lesquelles on ne souhaite pas essayer de résoudre l'équation, soit parce que ce ne sera jamais utile, soit parce qu'il est évident qu'il n'existe pas de solution simple ou intéressante. De manière générale, plus il y a de façons locales de résoudre une équation, plus Spark est capable de trouver une solution efficace du système global.

Par ailleurs existent aussi des commandes pour la création automatique de macro-objets, d'objets dynamiques, de macros-objets dynamiques, la création d'un fichier de simulation, et d'entrée de paramètres.

Évaluation

Spark génère également un programme adapté spécifiquement au problème posé. L'optimisation atteinte par cette génération est une réduction du nombre de variables sur lesquelles l'algorithme de résolution peut itérer. Cela donne à l'utilisateur une plus grande liberté pour exprimer les équations.

Spark est entièrement basé sur des objets sous forme équationnelle, ce qui permet des optimisations. Des comportements avec discontinuités ou hystérèsis peuvent être décrits dans un objet sous forme de conditions dans lesquelles l'équation est applicable. Dans son état actuel, l'algorithme de résolution ne traite pas spécifiquement le problème des équations discontinues.

Bien que l'interface graphique ne soit pas encore disponible, le travail de l'utilisateur est déjà facilité par une bibliothèque d'objets existants et par la facilité qu'il y a à créer de nouveaux objets.

2.3.5 FET / ZOOM

L'environnement ZOOM (Zone Organized Optimal Modelling) et sa base mathématique, le FET (Formalisme d'Évolution par Transfert) sont en cours de développement par le CNRS, à Toulouse et à Orsay [9]. Le but du logiciel ZOOM est la simulation de systèmes complexes dans lesquels les divers processus qui entrent en jeu sont fortement couplés. En plus de la modularité nécessaire à la simulation d'un ensemble de modèles variés, le FET met l'accent sur les couplages entre les évolutions des différentes parties du système. Il est un cadre conceptuel qui permet de décrire le comportement du système dans le temps ou en régime permanent. Un mode mixte avec des parties dynamiques et des parties stationnaires est même possible.

À la différence de beaucoup d'autres environnements, ZOOM utilise une visibilité complète entre les composants et les connexions intérieures du système. L'algorithme interne demande une représentation linéaire (ou linéarisée) du problème.

Le formalisme d'évolution par transfert (FET)

Ce formalisme est développé depuis 10 ans pour la simulation des systèmes thermiques et a abouti à une maquette du logiciel ZOOM qui implémente ce formalisme. Les deux opérations fondamentales décrites dans le paragraphe sur l'approche systémique (§ 2.1.4) sont utilisées aussi ici : le découpage du système en différentes parties et puis le raccordement de ces parties pour retrouver le système global. Un découpage en plusieurs niveaux hiérarchisés est possible. Le raccordement entre les objets à un niveau donné est appelé une famille. Dans cet objectif, le FET utilise deux types d'objets fondamentaux :

- les *cellules* qui représentent les composants du système. On suppose que l'évolution de l'état d'une cellule α dépend de son état η_α et d'un vecteur contenant des information sur le couplage φ de la cellule avec son environnement. Le comportement de la cellule est décrit par l'équation :

$$\frac{\partial \eta_\alpha}{\partial t} = G_\alpha(\eta_\alpha, \varphi_\alpha, t)$$

- les *transferts*, qui représentent des interfaces physiques ou des processus de couplage entre les cellules. Les transferts sont définis comme des relations où la variable d'interface φ est exprimée en fonctions des états des cellules connectées par cette interface et éventuellement des autres transferts φ' connectés à ces mêmes cellules.

$$\varphi = f(\eta, \varphi', t)$$

  Ceci représente les équations de contraintes auxquelles sont soumises les variables de transfert.

Une des différences entre ces deux types d'objets est le fait que le modèle des cellules est un modèle d'évolution au cours du temps avec des variables d'état, alors que le modèle des transferts est une contrainte instantanée dans les interfaces. Les cellules possèdent une inertie physique qui fait dépendre leur évolution temporelle non seulement des transferts mutuels mais aussi de leur histoire propre. Quant à l'interface, c'est un phénomène sans épaisseur ; son état dépend instantanément de son environnement.

Pour calculer l'évolution du système construit par les deux types d'équations ci-dessus, on passe par une discrétisation du temps et une *linéarisation* de l'équation des cellules. Ceci donne

$$A_{\alpha\alpha}\delta\eta_\alpha + B_{\alpha\varphi}\delta\varphi = \Gamma_\alpha \delta t$$

$$\delta t \sum_\alpha C^+_{\varphi\alpha}\delta\eta_\alpha - (1 + D_{\varphi\varphi})\delta\varphi = \Omega_\varphi \delta t$$

où α représente l'ensemble de toutes les cellules du modèle et $\delta\varphi$ l'évolution de toutes les variables de transfert au cours du pas de temps. La matrice C^+ décrit l'influence des cellules sur un transfert. La résolution dans le cadre du FET passe par l'élimination des variables d'état $\delta\eta_\alpha$ et on peut écrire

$$\delta\eta_\alpha = \delta\eta_{\alpha,\mathrm{dec}} + \mathcal{F}_{\alpha\varphi}\delta\varphi$$

$$(1 + D_{\varphi\varphi} - \delta t \sum_\alpha C^+_{\varphi\alpha}\mathcal{F}_{\alpha\varphi})\delta\varphi = \delta t(\sum_\alpha C^+_{\varphi\alpha}\delta\eta_{\alpha,\mathrm{dec}} - \Omega_\varphi)$$

Ici $\delta\eta_{\alpha,\mathrm{dec}} = A^{-1}_{\alpha\alpha}\Gamma_\alpha\delta t$ représente l'évolution spontanée qu'aurait la cellule si elle était découplée de son environnement (c'est à dire de ces transferts). La matrice $\mathcal{F}_{\alpha\varphi} = A^{-1}_{\alpha\alpha}B_{\alpha\varphi}$ décrit l'influence d'une évolution des transferts sur l'évolution de la cellule α. L'influence directe des autres transferts sur un transfert est contenue dans la matrice $D_{\varphi\varphi}$.

Par élimination des variables d'état on obtient ensuite l'équation-clef:

$$(1 + D_{\varphi\varphi} - \delta t \sum_\alpha C^+_{\varphi\alpha}\mathcal{F}_{\alpha\varphi})\delta\varphi = \delta\varphi_{\mathrm{ins}}$$

avec laquelle on peut calculer l'évolution $\delta\varphi$ des variables de transfert au cours du pas de temps δt. La variable $\varphi_{\mathrm{ins}} = \delta t(\sum_\alpha C^+_{\varphi\alpha}\delta\eta_{\alpha,\mathrm{dec}} - \Omega_\varphi)$ est l'évolution insensible des transferts. Elle indique la variation qu'auraient les transferts si les cellules suivaient leur évolution découplée. L'expression entre parenthèses à gauche de l'équation est appelée la matrice de couplage. L'analyse de cette matrice permet de repérer les transferts que l'on veut plus particulièrement observer.

Un autre aspect de ce schéma est la réduction du nombre des variables. Cette réduction se fait sélectivement de manière à faire ressortir les transferts que l'on veut analyser. La méthode consiste à regrouper les cellules de la partition initiale du système en différents sous-systèmes qui sont les *familles*. Ce regroupement en familles emboîtées donne une structure arborescente du système (cf. 4.2).

La procédure de résolution, appelée la *Navette*, consiste à parcourir la structure hiérarchique des familles et des cellules afin d'éliminer les variables internes

des familles qui se trouvent au niveau le plus bas, jusqu'au calcul des transferts qui se trouvent tout en haut de la structure. À chaque étape d'élimination, la matrice de couplage est modifiée et le vecteur de variables considérées est réduit. On substitue ensuite les valeurs obtenues par l'équations ci-dessus en redescendant jusqu'au niveau des cellules.

On peut trouver une description plus détaillée de la technique du formalisme FET dans [83], [84], et [82].

ZOOM

L'environnement logiciel qui implémente ce formalisme de résolution est appelé ZOOM. Sa structure a été développée pour rendre le FET utilisable en l'appliquant aux problèmes thermiques, dans un objectif de recherche.

Pour les deux types fondamentaux d'objets, il y a dans l'implémentation informatique ZOOM deux types de processeurs génériques correspondants : les *processeurs d'objets* et les *processeurs de transferts*. Les *processeurs d'objets* calculent l'équation

$$\delta\eta = \delta\eta_{\text{dec}} + \mathcal{F}\delta\varphi$$

en fonction du pas de temps δt, l'état η_0 et les transferts φ_0 au début du pas de temps. Les résultats sont $\delta\eta_{\text{dec}}$ qui décrit l'évolution de la cellule si elle était découplée de son environnement et la matrice $\mathcal{F}$ qui représente l'influence des transferts sur la cellule.

Les processeurs de transferts prennent comme entrées les mêmes valeurs que les processeurs de cellules : δt,η_0, et φ_0. Ils traitent l'équation

$$\varphi = f(\{\eta_{\alpha 0}\}, \{\varphi_0\})$$

Ses résultats sont les valeurs de transferts φ, la matrice $\boldsymbol{C}^+$ qui décrit l'influence des cellules sur le transfert et la matrice $\boldsymbol{D}$ qui décrit l'influence directe des autres transferts couplés. Il est possible d'utiliser un pas de temps variable que l'utilisateur peut spécifier. Des développements sont en cours pour permettre aussi des pas de temps non-homogènes dans les différentes parties du système. Ceci donne la possibilité de séparer les objets en familles pour profiter des calculs adaptés à des constantes de temps spécifiques.

Il existe divers langages descriptifs (voir fig. 2.9) associés aux différents niveaux d'abstraction de l'approche FET[5]. Pour l'instant, il y a six langages plus ou moins rattachés à trois niveaux d'abstraction.

La structure du système, c'est à dire le partitionnement et raccordement, sera décrite par le ZDL (Zoom Design Language). Les processeurs correspondant à un modèle physique sont décrits par le PDL (Processor Design Language). L'utilisateur ne devrait pas avoir de contact avec les autres langages qui sont plutôt des utilitaires pour un traitement mathématique (HMB, constructeur

5. Il n'est pas étonnant de retrouver ici une approche et notation similaire de l'approche SYMBOL, car les deux projets ont bénéficié des développements respectifs de l'autre.

physique	ZDL	PDL	
mathématique		HMB	
informatique		ZBM	ZTM
		ALLIS	

FIG. 2.9 - *les langages de ZOOM aux différents niveaux d'abstraction*

de matrices Hessiennes) ou pour l'aide à la programmation (ALLIS, Ada like language, ZBM, Zoom Bank Manager). L'ensemble de l'environnement ZOOM est implémenté en FORTRAN sur des machines UNIX. Un gestionnaire de base de données (ZEBRA) est par ailleurs mis à profit pour l'environnement ZOOM.

Plusieurs exemples ont été traités et comparés avec d'autres environnements de simulation. Les résultats montrent une fiabilité au niveau de la précision des résultats obtenus et une souplesse au niveau de l'interaction avec l'utilisateur.

Évaluation

La description des modèles d'une part et la description d'un système concret sont bien formalisées et structurées dans le cadre du FET et ZOOM. Des langages spécifiquement développés aident l'utilisateur à effectuer cette tâche, quelque fois un peu lourde. On trouve aujourd'hui une bonne bibliothèque de modèles existants.

La documentation sur cette approche particulière est une grande collection de brochures et d'articles. Comme l'idée principale de cette méthode est un peu compliquée, des cours de formation sont offerts par les laboratoires impliqués dans le projet.

L'environnement de logiciels ZOOM est très souple en ce qui concerne l'interaction avec et les modifications faites par l'utilisateur. ZOOM est strictement hiérarchisé, et cela pour deux raisons. D'une part, le système est décrit par découpage en morceaux de plus en plus petits qui donnent une structure d'arbre pour la représentation. D'autre part, il existe une distinction entre différentes couches d'abstraction pour la description des modèles. Ceci permet à l'utilisateur de travailler avec des notations qui lui sont plus familières comme une pompe, un tuyau, plutôt que des équations mathématiques.

Depuis quelques temps on peut effectuer une simulation à pas de temps variable ou la durée du pas est spécifiée par l'utilisateur.

2.3.6 Quelques autres environnements

Il existe bien d'autres environnements qui se situent dans le domaine de la thermique que nous allons brièvement mentionner :

ACSL est un produit qui existe maintenant depuis une bonne vingtaine d'années. Similaire à Neptunix, ACSL accepte un langage de description pour

ensuite générer un programme FORTRAN qui effectue la simulation. Ce langage est basé sur une définition de langages de simulation. Le sigle ACSL veut d'ailleurs dire Advanced Continuous Simulation Language.

ACSL est particulièrement adapté aux systèmes continus avec paramètres concentrés (*lumped parameters*). Mais il n'a pas l'avantage de Neptunix qui gère les discontinuités.

IDA (ou anciennement MODSIM) est un environnement de modélisation graphique et interactif qui est en cours de développement par le *Swedish Institut of Applied Mathematics* [71] [70]. Le noyau central de IDA est un solveur numérique qui est basé sur l'algorithme de GEAR sous sa forme DASSL [63]. Ce solveur peut traiter de grands systèmes d'équations algébriques et différentielles non-orientées. Pendant la simulation un pas de temps variable est utilisé ; il prend en compte les discontinuités et les états discrets ainsi que les phénomènes d'hystérèsis. Le format pour la représentation de modèles sur ordinateur est le *Neutral Model Format* [74], développé en coopération avec l'équipe Spark, où il peut être utilisé comme un préprocesseur supplémentaire pour la description de modèle (voir § 2.3.4 en haut). Le NMF permet une description des modèles indépendants des environnements. L'interface graphique de IDA facilite la saisie des connexions [15]. La compatibilité entre les types de variables est vérifiée et la hiérarchie de description est mise à plat.

CLIM 2000 est développé par *Électricité De France* depuis 1985 [40]. Conçu comme un outil de recherche, il est destiné à une large gamme d'applications concrètes comme l'évaluation énergétique et financière des projets, l'analyse de confort, le développement de système de contrôle. Pour cet objectif on utilise le concept d'une stricte séparation entre une bibliothèque de modèles et un solveur standard pour les calculs numériques (ASTEC [46]). Des études pour utiliser un autre solveur sont en cours.

Un élément de la bibliothèque comprend toutes les informations nécessaires pour son traitement et son installation. Il est décrit à l'aide d'un PROFORMA (voir §. 7.3.1). Un composant se présente sur l'écran graphique à l'utilisateur sous la forme d'une icône avec des pattes de connexion. Depuis 1989, le logiciel est opérationnel et on cherche à alimenter la bibliothèque. Dans ce but, a été mis en place une méthode de modélisation basée sur la distinction entre trois points de vue qui sont ceux d'un *utilisateur*, d'un *modélisateur*, et d'un *développeur de logiciels* [68].

2.4 Problèmes courants

La complexité des problème étudiés implique aussi une certaine complexité au niveau de la simulation. C'est pourquoi on a pu constater en thermique du bâtiment le développement de programmes volumineux. Les premières approches ont créé des programmes statiques d'une taille importante, souvent

limités à l'étude d'aspects spéciaux. Des problèmes qui étaient proches ou similaires n'étaient pas pris en compte pendant le développement du code initial, et demandaient chaque fois de nouveaux développements. L'inconvénient de cette approche est le manque de souplesse du code. Une modification localisée entraîne d'autres modifications éparpillées dans le programme. Le programme est difficile à corriger en cas d'erreur.

Bien que résolus dans certains environnement de simulation, quelques problèmes restent d'actualité :

- un premier point est la description pertinente non ambigüe de modèles. Ceci concerne aussi bien la définition d'un modèle élémentaire et composé, que sa représentation informatique des modèles. C'est un problème qui s'aggrave encore, lorsque l'on veut construire des bibliothèques de modèles et si l'on veut permettre l'échange de modèles entre différentes applications.

  Il existe des propositions pour les deux aspects de ce problème. Une première initiative consiste à normaliser la description avec les PROFORMAS [28]. Nous y revenons dans le chapitre 7.3.1. La deuxième approche est plus liée à l'exploitation informatique. Un format uniforme peut être le Neutral Model Format (NMF), qui est proposé par les développeurs du simulateur IDA.

- un autre problème qui est partiellement lié au point précédent, est la saisie du système sur ordinateur. La saisie comprend la structure du système et les paramètres. Tous les laboratoires qui ont développés les environnements recensés, développent finalement aussi une interface graphique pour faciliter la saisie. Exceptés TUTSIM et le préprocesseur ALLAN qui sont des produits commercialisés, aucun environnement ne supporte encore cette interaction graphique avec l'utilisateur.

- On peut constater des progrès constants de la qualité numérique de la résolution des grands systèmes d'équations (différentielles ou non). Mais la convergence des calculs de simulation n'est toujours pas assurée dans tous les cas. La gestion des discontinuités dans les équations et un mode mixte des simulation *temps discret – événements discrets* reste un des grands problèmes numériques.

- Bien que la puissance des ordinateurs ait atteint un niveau considérable pour des prix tout à fait raisonnables, la durée des simulations peut être trop longue. On peut diminuer le temps de simulation en faisant travailler plusieurs processeurs sur un même problème. Les environnements de simulations traités ici n'utilisent pas (encore) un possible parallélisme dans l'architecture des machine ou dans un réseau d'ordinateurs interconnectés.

- Un point intéressant pour la simulation est une évolution de la structure au cour de la simulation. Comment peut-on représenter de nouvelles

connexions qui s'établissent dans le temps? Comment peut-on représenter l'état d'un module d'une manière neutre pour qu'au prochain pas de temps ce module puisse être simulé par un modèle différent?

2.5 Le simulateur du projet SYMBOL : Motor-2

À l'intérieur du groupe GISE a été développé le projet SYMBOL pour notre propre environnement de simulation. Cet environnement reprend quelques idées déjà présentes dans les environnements présentés en haut. Mais nous nous centrons encore plus sur l'aspect de la réutilisabilité des parties de travaux effectués pour des études précédentes. Cela veut dire que les efforts mis dans la compréhension des phénomènes physiques, la modélisation, l'implémentation informatique ne sont pas perdus, mais peuvent éventuellement être repris dans une nouvelle étude d'un système thermique. Une méthode a été mise au point pour systématiser la description tant des objets réels que des modèles utilisés et de la représentation sur ordinateur. Elle nous permet de bien distinguer les différentes phases et niveaux d'une étude de comportement thermique. Les aspects de modularité pour des raisons de flexibilité et souplesse ont été prépondérants dans la conception du projet.

Parmi d'autres utilitaires de l'environnement SYMBOL, se trouve le programme de simulation Motor-2. Deux points importants le distinguent des simulateurs habituels. Une description hiérarchisée du système est maintenue pour la simulation. On ne met pas à plat toutes les connexion, mais on garde un partitionnement du découpage pour la résolution numérique. D'autre part, des mesures sont prises pour profiter du parallélisme éventuel, d'une part de l'architecture des ordinateur ou d'autre part, d'un réseau d'ordinateur pour diminuer les temps de calculs.

Dans le chapitre suivant nous allons expliquer et présenter plus en détail les idées de base du projet SYMBOL et du simulateur Motor-2.

Chapitre 3

Objectifs de Motor-2

3.1 Un simulateur « idéal »

Compte tenu des problèmes mentionnés dans le chapitre précédent, on peut tenter d'imaginer un environnement de simulation idéal. Cet environnement devrait soutenir toutes les phases d'une étude de système comme nous les avons présentées. L'objectif d'un outil logiciel est d'alléger et rendre plus efficace le travail d'un ingénieur de simulation. Ce programme doit couvrir les deux phases principales, la modélisation et la simulation.

Pour la phase de modélisation, l'environnement doit permettre de développer de nouveaux modèles et de gérer les modèles existants. Les modèles existants peuvent aider à la création de nouveaux modèles de deux manières. D'une part, ils peuvent servir comme patron pour le développement de modèles entièrement neufs. D'autre part, ils peuvent être des points de départ pour les modèles qui ne se distinguent que peu et qui sont dérivés des modèles existants. Sans trop restreindre la liberté de formuler l'algorithme pour le nouveau modèle, l'environnement peut proposer un cadre pour la représentation des modèles. Nous avions déjà mentionné les PROFORMAS qui poursuivent cet objectif. Des outils logiciels vérifient l'adhésion au cadre prescrit et la cohérence interne d'un modèle. À titre d'exemple, ils peuvent tester les unités des variables utilisées dans les équations (voir [7, chap. 2.7]).

Quant à la deuxième phase, la simulation, c'est une application classique de l'ordinateur. Comme nous l'avons vu au chapitre précédent, il y a déjà beaucoup de logiciels qui traitent cette phase.

Un bon environnement de simulation peut guider l'utilisateur dans son choix de modèles pour le problème donné. La description du système étudié doit être à la fois facile, pour garder une bonne vue d'ensemble, et souple, pour pouvoir facilement modifier une configuration en vue de comparaisons. Un mécanisme de contrôle peut vérifier les connexions entre les modules, la cohérence des modèles appliqués et la cohérence entre les modèles et les connexions.

Pour la résolution numérique du système, on peut imaginer un logiciel qui prépare les modules et leurs connexions d'un manière à rendre les calculs qui

suivent les plus efficaces possible. Ces calculs sont effectués pour résoudre le système. L'environnement doit donc proposer un algorithme de résolution qui soit rapide et adapté aux types de modèles. Beaucoup de difficultés numériques peuvent se présenter pendant la résolution. Un environnement idéal doit les reconnaître et aider à les surmonter.

Le temps d'exécution d'une simulation reste toujours un problème ouvert. Un utilisateur veut toujours ses résultats le plus vite possible. Le programme idéal de simulation est portable sur différents types de machines et il est rapide.

Aujourd'hui, les interfaces graphiques sont devenues indispensables. À toute phase d'interaction de l'utilisateur avec l'environnement logiciel, il faut une interface facilement compréhensible, que l'on peut piloter par des actions simples comme « *cliquer* » avec le bouton d'une «*souris* ». Ceci concerne aussi bien le développement de nouveaux modèles, la saisie de la structure du système étudié et des paramètres des modules utilisés que le traitement des résultats comme la présentation des graphes et des courbes.

Il y a quelques années, le projet SYMBOL a été formalisé et mis en chantier dans le groupe GISE. Parmi quelques autres objectifs, on y trouve aussi le développement d'un environnement de simulation destiné aux problèmes thermiques de bâtiment.

3.2 Cadre : le projet SYMBOL

3.2.1 Présentation générale

Les principales applications du projet SYMBOL concernent les systèmes complexes de grande taille, comme les bâtiments. Les objectifs de ces travaux sont premièrement l'amélioration de la connaissance des systèmes thermiques par la modélisation et l'analyse, et deuxièmement l'offre d'un ensemble modulaire et cohérent d'outils logiciels. SYMBOL est un acronyme qui signifie SYnthèse Modale et Boîte à Outils Logiciels et évoque les deux axes principaux du projet.

L'*analyse* et la *synthèse modale* reposent sur le fait qu'un système thermique linéaire peut être représenté avec précision par un modèle modal d'ordre faible. On peut obtenir un tel modèle par diagonalisation et réduction d'un modèle numérique discret détaillé, par réduction directe d'un modèle analytique ou par des méthodes heuristiques ([7, chap. 4], [62]). Si le couplage entre des sous-systèmes est linéaire aussi, la *synthèse modale* permet de trouver directement le modèle modal de l'ensemble en fonction des modèles réduits des sous-systèmes (voir [36]).

Le deuxième axe du projet SYMBOL concerne la « Boîte à Outils Logiciels ». L'apparition des nouveaux concepts logiciels et outils de programmation ont initié cette nouvelle approche de la modélisation dans le projet. Un des objectifs du projet SYMBOL est de mettre à dispositions des ingénieurs thermiciens ces nouveaux moyens. On se place à leur niveau pour la présentation des outils logiciels

comme des codes de calcul et des logiciels qui seront développés dans ce cadre. Ces outils doivent se présenter au thermicien d'une façon naturelle et conforme à son environnement habituel. Leur intérêt est d'alléger la tâche de l'ingénieur en évitant les nécessités habituelles d'un travail sur ordinateur. On veut lui épargner les questions qui se posent normalement au physicien théorique, à l'informaticien, au mathématicien ou au numéricien. C'est la *capitalisation* des connaissances sous forme de programmes et données informatiques.

Ces objectifs généraux du deuxième axe du projets SYMBOL sont abordés à l'aide des idées de la *modularité* et de la *visibilité*. Modularité et visibilité sont appliquées à travers toutes les phases d'une étude systémique mentionnées précédemment (§ 2.1.4).

Modularité

Parmi les conclusions que l'on peut tirer du chapitre précédent, on voit qu'il est souhaitable de pouvoir connecter facilement des morceaux de programmes afin d'en créer de nouveaux. Ces morceaux peuvent provenir de sources différentes, et peuvent être partagés entre plusieurs études. Des conventions relatives à la forme des programmes développés dans le projet SYMBOL ont conduit à une architecture informatique modulaire dans laquelle on peut connecter et substituer des modules écrits pour des applications *a priori* différentes (voir plus loin § 3.2.2). On trouve dans l'environnement SYMBOL d'une part des modules plus ou moins généraux comme par exemples les descriptions de composants d'un système qui sont dans un format précis. Ces descriptions ne sont pas spécifiques à une application particulière. Elles peuvent éventuellement être réemployés pour d'autres applications. D'autre part, il existe dans SYMBOL des modules qui sont spécifiques à certaines tâches, c'est à dire, adaptés à un contexte particulier. Les fichiers décrivant les modes de simulation pour le logiciel Motor-2 en sont un exemple.

Cette modularité encourage fortement une réutilisation des morceaux existants. Cette utilisation fréquente des modules induit des modules plus sûrs, mieux conçus, mieux documentés et surtout validés grâce à un plus important effort initial et continu globalement sensible. [1]

Visibilité limitée

Parmi les objectifs du projet SYMBOL se trouve avant tout la mise à disposition des résultats de recherche appliquée au monde des ingénieurs thermiciens. Pour diffuser les méthodes et algorithmes développés par les chercheurs, il faut souvent les réécrire et les remettre en forme. Au cours des travaux de recherche ceci a souvent déjà été fait, au moins en partie. Notre object est de réutiliser

1. Il faut remarquer que l'amélioration des modèles ne peut pas être garantie par une utilisation plus fréquente. Certaines modifications qui sont une « amélioration » pour un modèle donné, peuvent poser des problèmes dans un autre contexte. La validation et l'amélioration dépendent fortement de l'environnement d'usage.

directement ces résultats déjà existants. La séparation des rôles d'un utilisateur et d'un développeur de méthodes dans la conception du projet SYMBOL et Motor-2 a pour but de découpler les démarches et de faciliter la transmission de connaissances. Un des moyens pour l'atteindre est la notion de *visibilité*. Il existe des règles pour spécifier quelle est la part d'un module qui peut être vue par les autres. C'est une sorte d'interface qui définit d'une manière abstraite le comportement du module, une *spécification de ce qu'il fait*. Seule cette partie est accessible par le reste de l'environnement. L'implémentation de ces spécifications, *comment il fait*, est totalement cachée. Des modifications éventuelles dans cette partie n'ont pas de conséquences (directes) sur le continu des autres modules. Cette séparation entre spécifications et implémentations traduit une visibilité limitée sur les modules. Elle augmente la modularité et permet des échanges de modules sans difficulté. Elle facilite en même temps la maintenance et le développement de modules, et de cette manière la diffusion de ces derniers.

Le concept de visibilité limitée est aussi utilisé dans la nouvelle approche de programmation *orientée-objet* que nous allons traiter plus loin (voir § 4.3). L'idée de séparation entre différentes couches descriptives sera reprise et élargie dans la présentation des niveaux d'abstraction (§ 4.1), mais d'abord nous allons regarder l'architecture globale du projet SYMBOL et ce qui concerne la « Boîte à Outils Logiciels ».

3.2.2 Structure de l'environnement SYMBOL

L'environnement SYMBOL comprend aujourd'hui plusieurs modules logiciels autonomes. La communication entre les programmes passe en général par des fichiers et est assurée par une syntaxe spéciale basée sur les idées de modularité et visibilité[2].

Si on regarde la figure 3.1, on peut constater que tous les programmes s'organisent autour de bibliothèques. Ceci souligne le rôle important que jouent ces bibliothèques dans le projet SYMBOL. La gestion d'un ensemble de modèles (mais éventuellement aussi de module) passe par une « modélothèque », une sorte de bibliothèque de modèles. Ici on stocke les informations relatives aux modèles. Plus particulièrement on spécifie la partie visible d'un modèle, son interface. Différentes implémentations pour une même spécification peuvent être disponibles. Comme dans un jeu de construction on peut choisir les briques pour les connecter et pour construire une représentation du système étudié.

Néanmoins on retrouve dans la conception du projet SYMBOL et dans son architecture (fig. 3.1) un chemin principal qui part d'une description de système, passe par un logiciel d'analyse ou de simulation pour finalement obtenir des résultats. Regardons cette figure plus en détail.

Au milieu se trouvent les bibliothèques pour les différents types d'éléments

2. une description complète de la structure des fichiers SYMBOL se trouve en B.1.

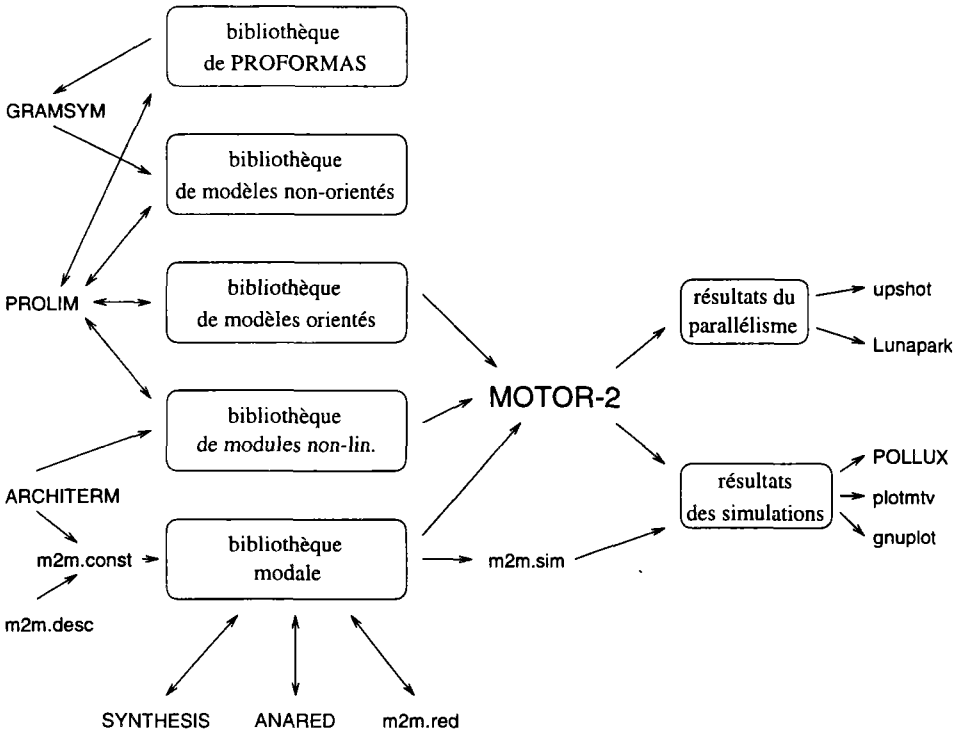


FIG. 3.1 - *La place de Motor-2 dans la structure du projet SYMBOL.*

du projet. Les interactions que l'on peut avoir avec une bibliothèque sont :

- l'interrogation de son contenu avec la possibilité de choisir un élément pour son application,
- des actions de maintenance comme ajouter, modifier ou enlever un élément,
- et finalement préparer et effectuer des transitions entre les bibliothèques. C'est à dire spécifier des paramètres et d'autres informations pour passer d'une bibliothèque générale à une plus spécifique.

Toutes ces actions sont des actions typiques d'une banque de données. Un prototype du gestionnaire de bibliothèques a été déjà développé sous le nom PROLIM (PROforma LIbrary Manager [30]). Une adaptation d'un programme de banque de données comme par exemple ORACLE remplacera ce gestionnaire à moyen terme. Puisque les bibliothèques sont encore très petites et le logiciel gestionnaire n'est pas réellement opérationnel, les actions sur les bibliothèques sont le plus souvent effectuées « à la main ».

Le deuxième grand bloc de modules logiciels concerne la description d'un problème. L'utilisateur saisit son projet en spécifiant les types et paramètres de ses modules. La tâche la plus importante pendant cette phase est la description des liens entre les modules. Nous reviendrons sur ce sujet. Une interface

graphique peut faciliter énormément ce travail de description. Un premier outil a été développé dans cette direction: c'est le logiciel ARCHITHERM. Sur la base d'un tracé en plan de chacun des niveaux d'un bâtiment, ARCHITHERM en produit une description structurée. La composition des composants et les caractéristiques thermophysiques des matériaux sont gérées sous forme de bibliothèques auxquelles ARCHITHERM fait référence. ARCHITHERM crée finalement les fichiers nécessaires pour le code suivant, actuellement m2m, les sorties pour Motor-2 et ALLAN sont en cours d'implémentation.

Beaucoup de logiciels existent déjà pour traiter des modèles modaux. La saisie peut passer par m2m.desc ou ARCHITHERM. m2m est un ensemble de blocs logiciels permettant de modéliser, analyser, réduire et simuler le comportement thermique dynamique d'un bâtiment. m2m.desc construit la représentation modale non-réduite d'un bâtiment décrit dans un fichier d'entrée. D'autres programmes travaillant sur des modèles modaux existent comme SYNTHESIS [36] pour la synthèse modale, m2m.red pour la réduction modale, ou ANARED [62] pour la création automatique d'un modèle modal réduit.

L'application principale pour la simulation au sein du projet SYMBOL est le logiciels Motor-2. C'est le seul moyen existant dans SYMBOL pour faire une simulation déterministe d'un système non-linéaire[3]. Motor-2 prend une description du projet et construit sa propre structure interne du système. Cette description, initialement abstraite, est traduite sous forme d'une représentation exécutable. De cette façon nous obtenons un noyau de la résolution indépendant des modules. Finalement, il exécute une simulation déterministe du comportement du système où chaque module peut avoir sa propre « file d'exécution »[4] (voir § 4.4 pour une discussion du parallélisme dans Motor-2). L'utilisateur peut choisir les variables à observer pendant la simulation. Différents formats existent pour stocker ces résultats.

Les utilitaires gnuplot et plotmtv du domaine public permettent de visualiser les résultats sous différentes représentations comme des courbes simples où des surfaces en 3D et colorées. Certaines configurations 3D peuvent être affichées à l'aide du programme Pollux, un logiciel développé dans le groupe GISE.

Les résultats qui concernent l'exécution de la simulation par plusieurs files indépendantes et parallèles dans Motor-2 peuvent être examinés par deux utilitaires. Lunapark affiche sur l'écran en temps réel les modules actifs. Il permet d'observer directement à tout moment quel module est traité par Motor-2. Comme Lunapark peut piloter la vitesse d'exécution de Motor-2, on ne peut pas examiner les durées relatives des différentes files d'exécution. Toutes les informations nécessaires pour une telle étude peuvent être écrites dans un fichier. L'utilitaire Upshot qui est habituellement utilisé pour les grandes machines massivement parallèles, peut le lire et afficher tous les instants de lancement, suspension, réactivation ou terminaison de toutes les files d'exécution.

3. un modèle linéaire ou linéarisé peut lui, être mis sous forme modale et utiliser le simulateur m2m.sim.

4. dans la littérature anglophone on emploi habituellement les termes *thread* ou *task*.

3.2.3 Contraintes pour Motor-2

Le cadre SYMBOL impose un cadre pour le développement d'un programme de simulation qui veut s'y intégrer. L'environnement de simulation doit soutenir les mêmes objectifs et idées que le projet SYMBOL. Il doit contribuer à un environnement qui se veut facilement maîtrisable pour un ingénieur thermicien ; usage facile, fichiers d'entrée et de configuration compréhensibles. Les concepts de la modularité et d'une visibilité limitée doivent être respectés. Le programme de simulation Motor-2 accepte la représentation standard des modules dans l'environnement SYMBOL. Motor-2 peut effectuer des simulation d'un système en connaissant seulement les spécifications des modules. (bien entendu, les implémentations des modèles utilisés doivent être disponibles.)

À part les contraintes imposées par la conception de l'environnement SYMBOL, il existait aussi des conditions organisatoires. Pour rendre effective la capitalisation des morceaux de programmes, la décision a été prise d'utiliser le langage Ada. Ce langage permet une conception *orientée objet* sur laquelle nous revenons dans le chapitre 4.3. Ada est un langage qui soutient bien le travail en équipe et la réutilisation, car on trouve une traduction directe de la visibilité dans les paquetages Ada.

Au niveau matériel, le laboratoire GISE dispose surtout des stations de travail Sparc. Le développement de Motor-2 a eu lieu sur ces machines et le système d'exploitation Unix. Néanmoins le programme de simulation tourne aussi sur des PC.

Après cette définition du cadre de travail, nous présentons une vue globale de l'environnement Motor-2 avec une initiation aux concepts de base. Ces derniers seront approfondis dans chapitre suivant.

3.3 Le simulateur Motor-2

Le logiciel Motor-2 est un programme de simulation basé sur une structure modulaire. Il est surtout conçu pour des simulations de modules continus dont le comportement dépend du temps.

Le logiciel actuel a été précédé d'une maquette qui a été développée au début du travail. Nous rappelons quelques détails de cette maquette qui pourrons nous être utiles dans ce qui suit.

3.3.1 Historique

Déjà dans la première version, appelée MOTOR (la version actuelle s'appelle Motor-2), nous travaillions avec une description du système qui était basée sur un découpage hiérarchique (voir plus loin et le chapitre 4.2 pour une discussion détaillée de ce concept). Un arbre de décomposition d'un système introduit la modularité dans la description du problème.

Dans le souci de créer une image informatique la plus proche possible de la réalité, chaque objet du système modélisé est représenté par une structure informatique équivalente. Dans le cas de MOTOR, nous utilisions des paquetage Ada pour implanter une visibilité limitée. La partie des spécifications Ada précise les parties visibles du module. MOTOR est un programme spécifique qui prend la description arborescente du système et qui génère des fichiers contenant le code Ada d'un programme qui simule le comportement du système. Pour chaque système, MOTOR crée un code de simulation qui est une image fidèle du système physique avec les données et méthodes nécessaires[5].

La technique numérique utilisée est une méthode de relaxation. Elle est liée à un raccordement entre deux modules qui se base sur l'égalité des températures et compatibilité des flux de chaleur. On envoie des nouvelles valeurs des températures de connexion aux modules connectés. Les modules calculent leur nouvel état et le flux qu'ils reçoivent du raccordement. Le bilan des flux est effectué sur chaque connexion et MOTOR modifie la température jusqu'à ce que le bilan soit nul. La méthode utilisée nécessite que chaque module soit capable de calculer la dérivée du flux par rapport la température.

Limites de la maquette

L'approche de MOTOR était intéressante pour de premières expériences sur l'assemblage des modèles et la génération du code. Mais nous avons trouvé des limitations sévères à l'usage de MOTOR. Premièrement il est lié à un algorithme numérique particulier ; seul des raccordement température - flux de chaleur sous condition de DIRICHLET sont possibles. En plus, la méthode demande aux modèles des capacités spécifiques comme les calculs des dérivées. D'autre part, l'usage de MOTOR est rendu très lourd par la génération de code. Pour chaque composant du système un paquetage de procédure a été créé ce qui cause une multiplication des lignes de code. Une simple modification dans la structure ou un nouveau système que l'on voulait simuler nécessitait une nouvelle génération de code et une nouvelle compilation du programme.

3.3.2 Présentation de Motor-2

Dans la version actuelle, nous voulons également garder des unités informatiques qui soient des traductions directes des modules de notre système du monde technique. L'idée principale est d'avoir une représentation informatique qui est la plus proche possible d'une image de la réalité vue par un ingénieur thermicien. On peut dire que l'objet technique est encapsulé dans une structure informatique.

Comme déjà mentionné, l'analyse du système à simuler passe par une description structurée de celui-ci. On découpe récursivement le système en sous-systèmes jusqu'à ce que les sous-systèmes élémentaires ne soient plus découpables. C'est une approche naturelle et elle est utilisée – même si ce n'est pas

5. C'est l'origine du nom : MOdule generaTOR.

explicite – dans tous les environnements de simulation. Nous considérons chaque composant ainsi obtenu comme une *boîte noire*. Pris séparément, il est libéré des contraintes de son environnement, et n'a plus d'effet en retour sur les autres éléments du système. Nous appliquons une notion de visibilité et de raccordement aux modules. Pour cet objectif, nous utilisons nos définitions des *frontières* en ce qui concerne la communication des module et des *interfaces* en ce qui concerne le raccordement entre les modules.

Le découpage hiérarchique nous fournit un arbre de dépendances qui décrit les liens entre les modules. Afin que la collection de modules représente bien le comportement de l'objet réel dont le système est le modèle, il faut les *raccorder* en imposant les contraintes nécessaires. Pour une description pertinente d'un système il ne nous faut pas seulement des modèles et ses paramètres pour les différents composants, mais nous devons décrire également les types des connexions entre les composants. Une des particularités de Motor-2 est le fait que l'on garde cette structure arborescente dans une représentation interne pendant la simulation. La hiérarchie descriptive du système est gardée et exploitée pendant la résolution. L'algorithme de résolution ne travaille que sur les connexions locales dans un module composé.

Un des objectifs de l'environnement Motor-2 est une interface d'interaction avec l'utilisateur dans laquelle on peut travailler à un niveau technique sans s'occuper de la réalisation informatique interne. Ceci concerne la description du système, ses composants, mais aussi les connexions entre les composants. Dans ce but a été développé une structuration de la démarche pour la description des modules. Pendant la description du système à simuler, l'utilisateur associe un modèle existant à chacun des composants et des connexions.

Cette approche permet une grande modularité de notre environnement de simulation. Nous pouvons créer des bibliothèques de modèles de composant et de modèles de couplage. Ces bibliothèques se remplissent à chaque nouvelle étude. La description de modèle peut faire référence aux modèles existants pour exprimer une similarité et une dérivation entre les modèles.

Non seulement au niveau des modèles, mais aussi au niveau de l'implémentation, nous augmentons l'efficacité par une réutilisation de l'existant. L'approche de la *conception* et de la *programmation orientée objet* s'y prête particulièrement bien. Elle convient bien avec notre approche globale et elle aide à réaliser concrètement ces idées à l'aide d'un ordinateur. Cela nous donne un environnement « ouvert », c'est à dire extensible dans lequel l'utilisateur peut facilement ajouter et modifier les modèles des bibliothèques.

Pour mieux exploiter les ressources en puissance de calculs qui existent dans les réseaux d'ordinateur et dans les machines parallèles émergentes, nous associons à chaque composant du système son propre code exécutable et sa propre progression dans les calculs. On peut ainsi distribuer les calculs sur plusieurs processeurs et exécuter en parallèle des parties de la simulation.

Les quatre concepts fondamentaux qui ne sont que brièvement présentés dans ce paragraphe sont approfondis dans le chapitre suivant. Nous parlons

notamment de la description du système par *découpage hiérarchique* et des *différents niveaux d'abstraction*, de la réalisation sur ordinateur dans le projet Motor-2 par une *conception orientée objet* et son approche du *parallélisme*.

Chapitre 4

Principes de Motor-2

Nous allons présenter dans ce chapitre les concepts fondamentaux pour le simulateur Motor-2 et son environnement. En premier lieu se trouve une structure de description à plusieurs niveaux de la modélisation et de la simulation. Cette approche a été établie dans le cadre d'une collaboration avec ALMETH [29]. Nous abordons le problème de la simulation du point de vue d'un ingénieur thermicien qui veut obtenir des réponses pertinentes à son problème. L'idée de découper un système avant de « recoller » les morceaux par raccordement est une démarche naturelle. C'est un processus rationnel qui simplifie le travail habituel. Cette technique permet par ailleurs l'établissement de bibliothèques de composants.

La réalisation informatique des concepts présentés – notamment l'idée de découpage et de raccordement – se prête bien à une *conception orientée objet*(COO). Le nouveau paradigme de la programmation orientée objet est né il y a quelque temps de la nécessité de mieux structurer le développement de produits informatiques de grande taille et / ou de longue durée de vie. Quelques idées et méthodes de la COO sont similaires à celles de l'approche systémique présentée au chapitre 2, et nous l'avons utilisée dans le projet Motor-2.

Une importante préoccupation de la simulation est le temps. Ceci concerne les deux aspects du temps. Il peut désigner le temps simulé ou encore la durée pour effectuer la simulation. À chaque instant de la simulation, les états des différents composants doivent être proches de la réalité. Pour calculer un instant donné, le temps simulé doit être cohérent dans tous les objets du système. Ce choix peut être réalisé par une extension naturelle de la COO qui est l'affectation des files d'exécution séparées à chacun des objets. Nous sommes ainsi amenés à parler du calcul parallèle. En plus de l'avantage d'approcher l'évolution réelle des objets, le calcul parallèle nous permet une meilleure utilisation des ressources matérielles.

4.1 Différents niveaux d'abstraction

Au début de notre travail sur le projet SYMBOL, nous avonspu constater une certaine ambigüité des termes employés dans le milieu de la modélisation des systèmes thermiques. Un premier classement des termes a permis de mieux les définir et de les associer aux différents niveaux d'une description [31][1]. Des discussions – et plus spécialement celles qui ont eu lieu dans le cadre du groupement ALMETH pour la publication [29] – ont finalement permis de fixer une structure générale de description. Son application dans le cadre du projet SYMBOL sera présentée ici.

Une structure à quatre niveaux a été établie pour une description efficace du processus de la modélisation. Chaque niveau ou couche est appelé un « monde». La structure permet de distinguer entre le *monde technique*, le *monde physique*, le *monde mathématique* et finalement le *monde algorithmique*. Une cinquième couche a été identifiée, celle d'un *monde informatique*, mais elle ne fait pas partie de la structure hiérarchique que nous décrivons ici. Dans l'environnement Motor-2 et plus généralement dans le projet SYMBOL, on trouve des représentations pertinentes au niveau informatique (par des fichiers descriptifs, par exemple) pour un objet à chaque niveau d'abstraction. La couche informatique recouvre les quatre niveaux; elle est donc située à côté.

mondes dans SYMBOL		*sigle*	*nom* ALMETH
le monde technique	le monde	Φ	Technos
le monde physique	informatique	T	Théoros
le monde mathématique	A	M	Analogon
le monde algorithmique	Automaton	N	Algorithmon

TAB. 4.1 - *les quatre niveau d'abstraction*

Le monde technique En général, on procède à l'analyse d'un système dans le monde technique. Nous décrivons le système par des expressions du « monde réel »[2]. Une pièce est constituée des murs, fenêtres, sol, plafond, air, ... À ce niveau nous posons des questions techniques en spécifiant les éléments du système.

Le monde physique Sous la couche technique se trouve le niveau où nous travaillons avec la physique. Des phénomènes physiques comme la conduction à travers un mur, sont associés aux objets du monde technique. Ici se trouve la modélisation, c'est-à-dire la construction des modèles par une interprétation théorique du système technique. On spécifie les hypothèses

1. À l'époque nous croyions cette difficulté due à une maîtrise insuffisante de la langue française. Il s'est avéré plus tard que c'est aussi une difficulté inhérente au problème.

2. On pourrait placer le *un monde réel* au-dessus du monde technique. La perception de ce que nous voyons comme *réalité* est un sujet philosophique et il n'est pas traité dans ce travail.

et simplification sur l'objet technique. On définit les lois physiques régissant les phénomènes retenus. C'est également l'endroit où l'on doit définir l'échelle de l'analyse aboutissant à une granularité judicieuse.

Le découpage d'un système en composants et sous-système peut se prolonger ici. L'espace entre les façades constituant une pièce est un composant technique. En observant les phénomènes physiques nous pouvons maintenant distinguer entre la convection naturelle de l'air et un rayonnement thermique entre les faces de la pièce.

Le monde mathématique Dans la couche mathématique nous obtenons une description plus rigoureuse et formelle des lois. Il permet d'un côté des solutions analytiques, et de l'autre une compréhension qualitative des processus, des relations et de la structure du système. Le langage mathématique est un moyen d'exprimer d'une manière abstraite notre perception du monde réel.

Le monde algorithmique Pour des problèmes compliqués, il faut choisir des méthodes de calculs spécifiques. Nous décrivons maintenant des algorithmes correspondant à notre modèle mathématique pour trouver une solution numérique explicite à ce niveau (ce niveau est également appelé le *monde numérique* quelque fois). Si la formulation mathématique de la couche précédente demande l'inversion d'une matrice, c'est ici que l'on exprime précisément la méthode à employer.

Le monde informatique Comme déjà dit, nous ne voyons pas ce monde comme une couche supplémentaire de description, mais comme un support pour toutes les couches précédentes. Dans notre environnement d'outils logiciels de simulation, nous essayons de représenter tous les niveaux descriptifs par des moyens informatiques. La traduction d'un algorithme du monde numérique dans un langage de programmation est une des tâches les plus simples. Des représentations pertinentes pour les autres niveaux seront proposées dans les paragraphes suivants.

Il y a deux chemins associés à ces niveaux de description. Le premier va du haut vers le bas (*top-down*). C'est un processus qui part d'un objet réel (une pompe, par exemple), cherche un phénomène physique (compression du fluide), développe une analogie mathématique (loi de NEWTON), et trouve une méthode algorithmique de résolution. Le chemin inverse va du bas vers le haut (*bottom-up*). C'est la traduction et l'interprétation des résultats numériques aux niveaux mathématique et physique en espérant trouver des réponses aux questions techniques de départ.

Comme déjà mentionné, il existe plusieurs interprétations du mot *modélisation*. Nous reprenons notre définition du chapitre 2.1.1 qui distingue la phase de la création de modèles (*modélisation*), et la description du système (*simulation*). Avec cette définition, nous pouvons reconnaître conceptuellement deux chemins (ou flux d'information) complémentaires pour aboutir à une description pertinente de système. Ces deux chemins sont souvent parcourus tous les deux en même temps.

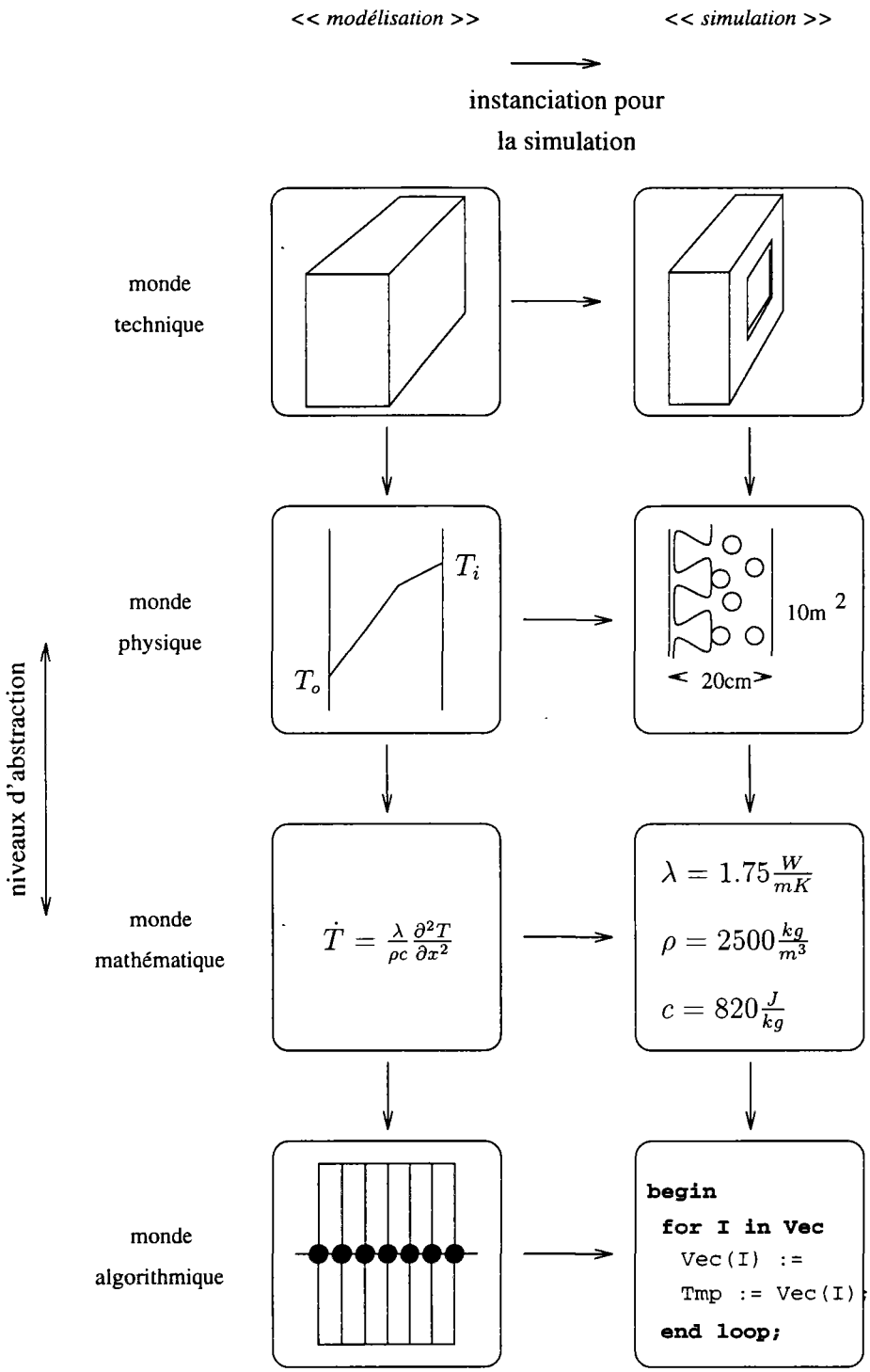


FIG. 4.1 - *Les transitions entre les couches d'abstraction.*

Nous remarquons que la transition de l'objet du monde réel vers un modèle dans les mondes physique et mathématique demande un effort d'abstraction. Un modèle est plus abstrait que l'objet réel ; il contient des hypothèses et des choix de simplification. Par contre, la couche algorithmique et plus particulièrement

la couche informatique contiennent des concrétisations du modèle pour effectuer la simulation. Ceci est lié aux deux chemins de la figure 4.1, à gauche les étapes d'une modélisation, à droite l'application pour la simulation. Les flèches indiquent des flux d'information. Pour arriver à un programme de simulation on a utilisé à la fois des informations qui viennent de la modélisation de gauche et des information sur l'objet concret du haut, c'est à dire d'une abstraction plus élevée.

Pour illustrer la structure des mondes sur laquelle s'effectue la description, nous nous limitons dans ce paragraphe aux modèles de connaissance qui sont une construction formelle déduite d'une théorie. Il est évident qu'il n'y a pas de représentation objective et incontestée de la réalité acceptée comme référence par tous. Cette absence de référence absolue conduit à la difficulté à distinguer l'objet-phénomène du modèle. Cela vient du fait qu'un objet-phénomène réel n'est jamais perçu qu'à travers des représentations mentales qui sont des modèles. En toute rigueur – nous l'avons déjà dit – il est admis qu'un modèle n'est jamais parfait et que la notion de pertinence est relative aux objectifs qui lui sont assignés et à l'utilisation qui en est faite.

Description aux quatre niveaux

Nous allons suivre l'évolution de la description d'un mur. Nous distinguons à chaque niveau entre les deux voies de la figure 4.1, le développement de modèle (*modélisation*) et son application au problème concret (*simulation*). Dans le chapitre 5.3.4 cet exemple est élargi au mur bicouche pour inclure la description du raccordement.

- Dans le *monde technique*, nous regardons un mur quelconque (« *modélisation*»). Son apparence concrète spécifique est le mur en béton de la façade nord du bâtiment (« *simulation*»). À ce niveau nous spécifions l'objet réel et le matériau de composition.

- La simplification de la modélisation s'effectue dans le *monde physique*. Nous voulons observer uniquement le comportement thermique du mur conducteur dans la direction de son épaisseur. Nous parlons d'une conduction monodimensionelle et d'une capacité thermique (« *modélisation* »). La géométrie du mur est fixée à une épaisseur de 20cm et une surface de 10 m^2 (« *simulation* »).

- Au niveau du *monde mathématique* la conduction est exprimée par l'équation connue (« *modélisation* ») :

$$\dot{T} = \frac{\lambda}{c\rho} \frac{\partial^2 T}{\partial x^2}$$

  Les propriétés thermo-physiques du matériau sont déterminées comme $\lambda_{\text{béton}} = 1,75 W/mK$, $\rho_{\text{béton}} = 2500 kg/m^3$, $c_{\text{béton}} = 820 J/kg$ (« *simulation* »).

- Afin de résoudre cette équation, nous devons choisir un *algorithme*. La discrétisation de l'espace, l'association des capacités aux nœuds créés et les résistances entre les nœuds nous donnent une représentation par différences finies. Cet ensemble d'équations différentielles ordinaires peut être résolu par un algorithme d'intégration (voir annexe D.2) qui progresse par pas de temps discrets (« *modélisation* »).

  Avec ces paramètres, nous pouvons choisir le nombre et l'emplacement des nœuds pour le module spécifique de l'objet étudié.

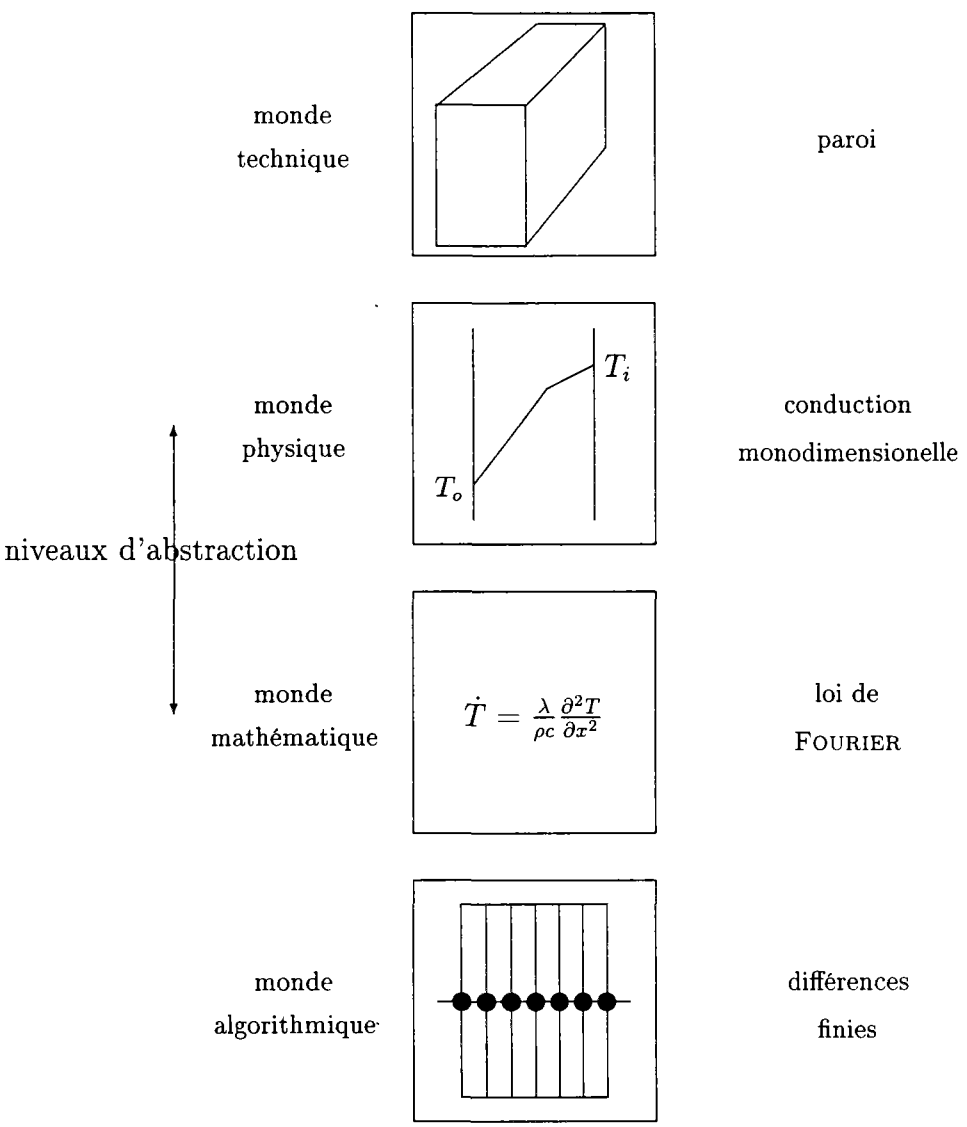


FIG. 4.2 - *Représentation d'un modèle aux différents niveaux d'abstraction.*

Autres approches comparables

Le projet CLIM 2000 d'EDF (voir page 32) utilise une approche similaire à la nôtre. On sépare les rôles de l'*utilisateur*, du *modélisateur*, et du *développeur de logiciels* [68]. Les similitudes avec nos couches d'abstraction sont évidentes. L'utilisateur décrit son système et fait la transition entre les mondes réel et physique. Le modélisateur développe des modèles en trouvant de bonnes représentations mathématiques pour les phénomènes physiques. Le développeur de logiciels est préoccupé par les aspects numériques et l'implémentation informatiques des modèles.

Dans l'environnement IDA, on peut trouver une « représentation des objets» (*object representation*) générale qui contient toutes les information sur un composant. Un gestionnaire (*panel handler*) extrait les informations nécessaires, une fois pour l'utilisateur en vue d'un affichage graphique et une fois pour le solveur afin de créer le système d'équations.

On voit aussi ailleurs émerger cette idée de séparation. Dans le domaine de la simulation et de la modélisation, on parle par exemple de la séparation entre des éléments physiques, des éléments de contrôle et ceux porteurs d'information [65].

Les différents niveaux d'abstraction aident à la description complète et surtout pertinente d'un système. Se rendre compte à quel niveau on se trouve à un moment donné, permet une meilleure organisation du travail et avec cela une meilleure maîtrise de la complexité. Le découpage d'un système en unités plus petites a le même objectif. La réalisation du découpage dans l'environnement SYMBOL et le simulateur Motor-2 montre quelques particularités.

4.2 Découpage hiérarchique

Il est raisonnable de décrire des systèmes compliqués par leurs sous-systèmes et les liens entre ces sous-systèmes. C'est une approche classique de type cartésien – découper de plus en plus pour mieux comprendre les différents éléments du système. On la trouve dans toutes les méthode d'analyse de système. L'approche *systémique* du chapitre 2.1.1 est complémentaire à ce découpage. Elle met à son tour l'accent sur les liens et les rapports entre les composants. Une décomposition d'un système peut être représentée par un graphe qui relie les composants.

Pour obtenir ce graphe nous coupons le système en sous-systèmes et ces derniers sont découpés à leur tour. De cette manière nous descendons récursivement dans le système jusqu'à l'obtention d'une granularité jugée suffisante. À chaque étape on peut choisir si l'on continue par une description encore plus fine, ou si l'on arrête. En découpant nous construisons notre graphe qui a sa racine en haut et des branches de plus en plus fines vers le bas et une hiérarchie des visibilités.

Prenons l'enveloppe du bâtiment comme exemple d'illustration. Au lieu de tenter d'élaborer directement un modèle thermique pour le bâtiment, on procède à un découpage hiérarchique. On commence par un découpage de l'immeuble en différents étages. Ensuite nous regardons étage par étage et nous coupons chacun d'entre eux en différentes pièces (cuisine, chambre, salle de bains, ...). Une pièce peut être décrite comme un ensemble composé des murs, de l'air, des fenêtres, etc. Et finalement nous allons distinguer les couches d'un mur (isolation, béton, bois). La construction du graphe (fig. 4.3) qui représente le bâtiment, se fait en même temps que le découpage.

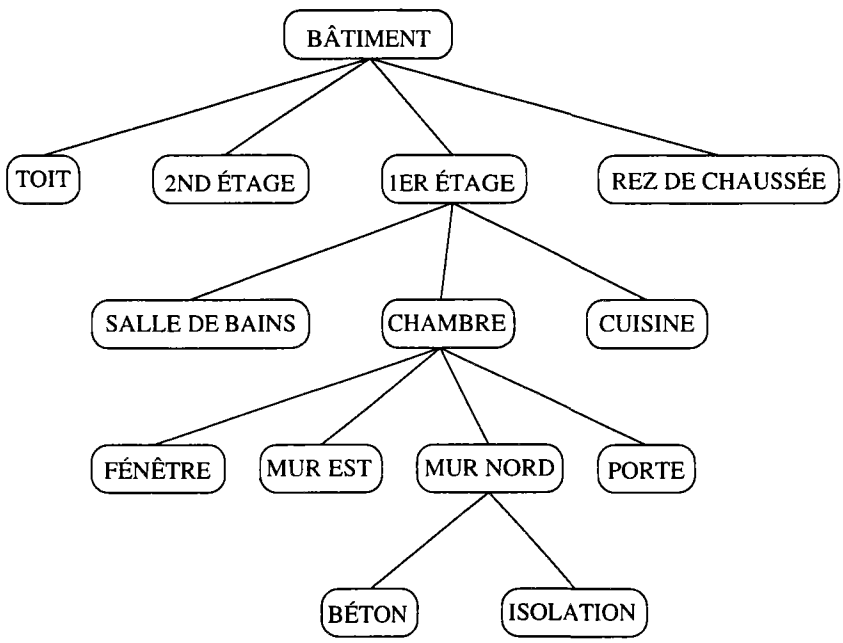


FIG. 4.3 - *un des choix possibles pour un graphe arborescent d'un bâtiment. C'est un arbre de profondeur 4 qui a dix feuilles terminales.*

Il n'y a certainement pas qu'une seule façon de découper un système. C'est le choix de l'utilisateur, et il dépend fortement de sa perception du système. Des graphes résultant du découpage d'un même système peuvent être très différents. Un point intéressant est l'influence du découpage sur la résolution numérique du système. En fonction du nombre de sous-modules et de niveaux de découpage, la simulation est plus ou moins longue : la vitesse des calculs avec le programme de simulation Motor-2 dépend de la description du système. Au chapitre 9 nous traitons quelques aspects de ce point et en déduisons des conséquences pour améliorer la construction du graphe. Par ailleurs, l'arbre n'est généralement pas un arbre au sens strict, car certaines feuilles appartiennent à plusieurs branches (le plafond du premier étage est aussi le plancher du deuxième étage). Il nous aurait intéressé de traiter un graphe général dans le projet Motor-2, mais malheureusement, ce n'est pas possible dans sa conception actuelle. Seules les déscription en structure strictement arborescentes peuvent être simulées par Motor-2.

Les composants que l'on ne peut plus (ou que l'on ne veut plus) découper

sont appelés des *composants terminaux*. Ils sont les feuilles du graphe arborescent. Les composants des niveaux supérieurs qui ont des branches sont appelés *composants composés*. Dans notre exemple CHAMBRE est un *composant composé* et BÉTON est un *composant terminal*.

Pour décrire des relations entre les niveaux, nous employons la terminologie des filiations. Un composant A est dit *père* du composant B, si A est décrit explicitement comme un assemblage de B avec d'autres composants. Dans ce cas le composant B est appelé le *fils* de A. De cette façon on peut dire que le MUR NORD est le *père* de l'ISOLATION et du BÉTON.

Ce découpage d'un système est une phase dite *top-down*, car on part du haut d'un système et on l'analyse vers le bas. L'existence et la nature des relations sont analysées par une deuxième phase. Cette phase inverse – dite *bottom-up* – reconnecte les composants du bas vers le haut. En rétablissant les liens entre les composants on reconstruit le système initial. Ce sont ces liens qui imposent des contraintes à l'évolution libre des composants de façon à ce qu'ils décrivent leur comportement dans le système. La simulation, c'est à dire, la résolution numérique du système doit respecter et résoudre les contraintes de raccordements.

Sans trop entrer dans les détails qui sont présentés dans le chapitre « raccordement » (§ 5.3), le travail essentiel du simulateur Motor-2 se trouve au niveau des modules composés. Comme nous voulons construire un environnement de simulation ouvert, nous ne nous occupons pas trop des algorithmes qui sont employés dans les modules terminaux. C'est la résolution des raccordement qui est la tâche principale de Motor-2. Nous installons entre les sous-modules des *interfaces* qui sont des points de calcul dans les modules composées. Par ailleurs, l'approche du simulateur Motor-2 est de garder dans la phase de la résolution la structure arborescente qui provient de la description du système. Nous trouvons ici un des points particuliers de Motor-2, une hiérarchie de résolution. Les résultats d'un module composé sont redistribués au niveau supérieur ou un autre module composé les reprend et les traite comme s'ils venaient d'un sous-module quelconque. Dans le chapitre « résolution » (§ 6) nous traitons les différents algorithmes qui peuvent être exploités par Motor-2.

La modularité que l'on obtient par le découpage – mais également par la description aux différents niveaux d'abstraction – se prête bien à la réalisation informatique, où nous retrouvons une équivalence entre les objets du monde réel et leur représentation informatique. Cela est l'idée de la programmation orientée objet qui est expliquée dans le paragraphe suivant.

4.3 Objectifs et principes de la *Conception Orientée Objet*

Afin de clarifier l'idée de la *conception orientée objet* (COO), nous allons citer quelques définitions que l'on peut trouver dans la littérature. MEYER[3] voit

3. B. MEYER est le développeur principal du langage Eiffel.

la conception orientée objet comme suit : « *la conception orientée objet peut être définie comme une technique qui, contrairement à la conception classique (fonctionnelle), base la décomposition modulaire du système sur les classes d'objets que le système manipule, et non sur les fonction que le système effectue.*[4] » [58].

BOOCH définit un objet : « *un objet est une entité qui a un état, qui est caractérisé par les actions qu'il subit et les actions qu'il demande aux autres objets. Un objet est une instance unique d'une classe d'objet. On associe toujours deux points de vue à l'objet : la vue extérieure qui définit l'interface, et la vue intérieure qui définit l'implémentation.* » [11].

Nous retrouvons dans ces deux définitions aussi les idées de l'approche systémique, notamment le découpage, l'identification des composants du système, et surtout la mise en relief d'une visibilité limitée qui masque la complexité, et des communication et dépendances entre les objets.

Des nombreux auteurs ont tenté de définir des critères de qualité pour un logiciel ([?], [11]). Ces qualités et les concepts qui les entourent justifient l'utilisation des techniques orientées objets, dans la mesure où ces techniques aident à la conception de logiciels dotés de qualités suivantes :

Réutilisabilité C'est l'objectif fondamental de la programmation orientée objet. Le transfert des solutions existantes sur un nouveau problème entraîne souvent, dans les langages conventionnels, la récapitulation des pas nécessaires et une connaissance en détail des fonctions et procédures. Dans ce cas, des modifications sont seulement possibles par des copies et des extensions importantes dans les textes sources (par exemple, en utilisant des compilations conditionnelles). Les langages orientés objet cherchent à rendre un programme réutilisable, en tout ou en partie, pour de nouvelles spécifications.

Souplesse Les efforts en vue de modifier un programme sont souvent plus importants que ceux qui sont nécessaires à sa création initiale. C'est souvent le cas pour les grands systèmes où les coûts de maintenance dépassent largement le coût de réalisation initiale. L'approche orientée objet facilite cette « maintenance » surtout pour les programmes complexes, grâce à des interfaces d'objets bien définies et l'élimination des effets de bord non-souhaités. Comme un système ne reste jamais fixé, les demandes faites au programme évoluent et il doit être facilement adaptable lors de l'évolution et de l'extension des spécifications.

Abstraction L'image du monde réel comme un ensemble d'objets autonomes qui communiquent par échange de messages permet la programmation aux différents niveaux d'abstraction. De cette façon on développe tôt des spécifications et ensuite des prototypes qui aident à éviter des travaux répétitifs et à structurer l'avancement. La gestion d'un seul niveau à la

4. *object oriented design may be defined as a technique which, unlike classical (functional) design, bases the modular decomposition of a [...] system on the classes of objects the system manipulates, not on the functions the system performs.*

fois facilite le travail du programmeur. Par un meilleur couplage entre spécification et implémentation, celui-ci devient capable de maîtriser des projets plus gros.

Ces structures et méthodes ont été développées dans différents langages pour réaliser ces objectifs de la COO au niveau de la programmation. Les concepts de l'interface de communication (la séparation entre les spécifications et l'implémentation d'un module), l'encapsulation de données (visibilité limitée) et l'héritage des classes existantes (extension de types) y sont les techniques de base.

Encapsulation et masquage d'information Un type de données abstrait caractérise un ensemble de valeurs et d'opérations applicables sur les objets de ce type. Les mécanismes de définition d'un type abstrait permettent au programmeur de spécifier le comportement d'un objet sans aucune référence à la représentation de l'objet, ni à l'implémentation des opérations définissant son comportement. Le programmeur dit ce qu'un objet fait, sans dire comment il le fait. Ce n'est bien sûr qu'une inversion chronologique ; il faudra bien à un moment ou un autre, décrire la méthode afin que le programme exécutable puisse être constitué. Les abstractions logiques doivent être protégées de telle façon que les implémentations ne puissent pas être directement referencées. On ne peut accéder au contenu d'un objet protégé que par ses méthodes ou les parties qu'il a rendues visibles. Pour éviter ces accès indésirables, deux mécanismes sont appliqués en même temps : encapsulation de données et masquage d'information.

- L'ensemble des fonctions qui peuvent interroger et modifier les données du type doivent être définies au moment de la création du type.
- La représentation interne d'un type doit être cachée à l'utilisateur de ce type. Seules les fonctions définies avec le type donnent accès à un objet pour lire et modifier les données. « La raison d'être du masquage est de rendre inaccessible certains détails qui ne devraient pas influencer d'autres parties du système » [12].

L'encapsulation de données et le masquage d'information relient de façon étroite les données et leurs opérations. Par conséquent, les fonctions capables de manipuler les données sont limitées à des régions localisées. Par ce mécanisme on obtient une réduction des liens entre les modules, car l'accès aux détails d'une implémentation est restreint. Pour accéder aux informations souhaitées, il faut passer par les fonctions dédiées. Ceci rend possible une modification interne (remplacement de la représentation ou de l'algorithme) sans dérangement pour les clients. L'encapsulation permet une séparation claire entre la définition d'une interface abstraite d'un type avec ses valeurs potentielles et son comportement, et l'implémentation, c'est à dire la réalisation, de ces fonctions. Les bénéfices de cette séparation sont nombreux : l'abstraction facilite la maintenance et la compréhension des systèmes en réduisant les détails que le développeur

doit prendre en compte simultanément, et en limitant la portée des changements. La fiabilité des systèmes s'en trouve augmentée suite à la portée limitée de certaines opérations [12].

La modularité Le concept de modularité n'est pas un concept récent. Il est appliqué d'une façon ou d'une autre dans l'ensemble des méthodologies. Une définition possible de la modularité peut être la suivante : les modules sont des composants élémentaires autonomes, qui permettent la réalisation d'un ensemble par combinaison.

Le concept d'objet et de classe Un objet pour un langage de programmation est une partie clairement délimitée du système modélisé. Il correspond donc à notre composant d'un *système général.* Chaque objet contient des information qui lui sont propres (par exemple l'objet `brûleur` connaît sa puissance), ou des informations sur les relations avec les autres objets. Le comportement de l'objet et la représentation de l'information sont encapsulés dans l'objet même. Le seul moyen d'obtenir ou de modifier des informations est d'effectuer les opérations sur l'objet.

Une classe est une implémentation d'un type abstrait de données. C'est donc un type défini par l'utilisateur (programmeur) avec une représentation interne de données et un jeu de fonctions s'appliquant sur ces données. Les valeurs qui sont propres aux objets d'une classe sont souvent appelées *attributs*, les fonctions qui agissent sur celles-ci sont appelées *méthodes*. Une classe est une structure générique avec laquelle on peut créer des objets. Elle contient donc une description générale dans le sens de l'encapsulation et du masquage de données pour tous les objets appartenant à cette classe. Chaque objet créé à partir de la classe est une instance unique de cette classe. C'est la classe qui spécifie les fonctions possibles sur les objet; mais c'est l'objet même qui contient l'information unique à son instance.

L'héritage et le polymorphisme Il est parfois difficile d'utiliser les abstractions de données ; une fois définies elles n'intéragissent pas avec le reste du programme, et il n'y a aucun moyen de les adapter à de nouvelles situations sans les redéfinir. Un mécanisme pour l'extension et la réutilisabilité est donc l'*héritage*. L'héritage permet de définir de nouvelles classes à partir de celles qui existent déjà. Une classe héritière est une extension de la classe dont elle hérite des méthodes et attributs. Ses spécificités sont précisées par le rajout de variables d'état, le rajout de nouvelles fonctions et la redéfinition de fonctions existantes. Une des particularités de l'héritage est que tous les objets d'une classe *fille* héritant de sa classe *mère* peuvent être considérés comme un objet de la classe mère même. Un objet peut être vu comme une instance d'une de ses classes mères par un module demandant une action. Mais l'objet réagit selon sa propre définition de la fonction et non celle de la classe mère. Ceci est appelé *polymorphisme*. Lorsqu'un objet reçoit un message (est appelé), il déclenche une recherche dynamique de la méthode à effectuer. Si la classe de l'objet ne possède pas

cette fonction, la classe mère est donc consultée, et ainsi de suite (*liaison dynamique*).

Les conception et programmation orientées objets sont aujourd'hui acceptées comme style préféré de construction de programme (on parle du *paradigme de programmation*). Nous l'avons appliquée à la construction du programme Motor-2. On peut remarquer les correspondances entre la conception orientée objet et l'approche systémique de description que nous employons dans le cadre de la simulation. Cette correspondance permet une transition directe entre les objets de l'environnement Motor-2 et leur implémentation sur ordinateur. Nous pouvons par exemple assigner à chaque module créé par le découpage son objet informatique correspondant. Tous ces objet appartiennent à la classe des modules. Cette classe spécifie la vue extérieure minimale des données et les actions que tout module doit être capable d'effectuer. L'application de la conception orientée objet dans le contexte du simulateur Motor-2 devient plus claire au prochain chapitre quand nous présentons plus en détail le concept de module.

Comme nous l'avons vu, l'approche orientée objet utilise l'image des objets indépendants qui communiquent par échange de messages. Cette idée se laisse facilement étendre à l'application d'un parallélisme réel entre les objets. La correspondance entre la réalité et la représentation informatique devient encore plus proche, car dans la réalité aussi, tous les phénomènes ont lieu simultanément.

4.4 Parallélisme

Quelle est l'idée de la parallélisation? Il existe beaucoup de problèmes de calcul dans des domaines différents qui ne peuvent être traités même sur les machines les plus rapides, simplement parce qu'elles sont encore trop lentes ou trop petites. On peut citer comme exemple les prévisions météorologiques[5], ou les simulations en mécanique de fluides. C'est aussi vrai dans la thermique où les modèles deviennent de plus en plus grands et demandent toujours plus de ressources de calcul. L'idée de la parallélisation pour accélérer les calculs est de réunir les puissances de plusieurs processeurs et de faire travailler ensemble plusieurs machines sur un même problème, plutôt que d'épuiser la puissance d'une seule machine. Depuis quelques années on peut constater l'émergence rapide des machines parallèles. En même temps, nous voyons dans les laboratoires de recherche et dans l'industrie des machines connectées entre elles dans un réseau. Ce réseau donne la possibilité aux machines de communiquer et ainsi, la possibilité de travailler ensemble sur un même problème. Nous allons donc présenter notre approche pour utiliser ces nouveaux moyens de calculs dans notre projet.

L'approche orientée objet est très liée à l'image d'objets différents qui communiquent par échange de messages. La programmation orientée objet est ainsi

5. dont la précision reste limitée par principe, même avec les machines les plus rapides. Malgré tout, les modèles deviennent de plus en plus détaillés et demandent plus de ressources de calculs.

une conception bien adaptée au parallélisme. Elle se présente comme extension naturelle de l'approche orientée objet. Souvent désigné par le sigle COOP (Concurrent Object Oriented Programming) [1], on peut employer trois voies différentes pour sa mise en œuvre :

1. **parallélisme caché** Un programme est conçu séquentiellement comme d'habitude. C'est la tâche d'un compilateur optimisant de découper automatiquement ce code en une partie séquentielle et une autre partie parallèle. Le programmeur ne s'en aperçoit pas, mais le programme peut utiliser les ressources d'une machine parallèle.

2. **synchronisation explicite** Comme extension des programmes orientés objet, on ajoute de nouvelles classes pour des objets actifs (processus, tâches). Ces objets ont virtuellement chacun son propre processeur. Seuls certains objets sont actives. Des mécanismes de contrôle assurent l'interaction entre les objets. Les objets actifs suivent eux-mêmes certaines conditions qui doivent être remplies pour l'exécution. La synchronisation est seulement partiellement centralisée.

3. **des systèmes pour communication asynchrone** Tous les objets sont actifs. Ils envoient des messages sans attendre la réponse, qui revient éventuellement à n'importe quel moment comme message à l'expéditeur. Le système entier, la pensée et les programmes résultants sont conçus pour la communication asynchrone. La synchronisation explicite est superflue dans la plupart des cas.

Architecture des ordinateurs parallèles

Dans un ordinateur traditionnel avec un seul processeur, ce dernier exécute un programme qui réside dans la mémoire. La suite des instructions parcourues s'appelle le *flot d'instructions.* Le microprocesseur opère sur des données qui se trouvent dans ses registres. La suite des valeurs de ces registres s'appelle *flot de données.* Les machines parallèles généralisent ce modèle de fonctionnement de deux façons, soit en multipliant le nombre de flots d'instructions, soit en multipliant le nombre de flots de données.

Si une machine a plusieurs flots de données et un seul flot d'instructions on parle de machine SIMD (*single instruction stream, multiple data stream*). Dans ce type de machines, les différents processeurs constituant la machine exécutent le même programme, mais sur des processeurs différents et des données différentes. Chaque instruction du programme à exécuter est envoyée de façon synchrone à tous les processeurs : on a donc un même flot d'instructions pour chaque processeur. Par contre, les mémoires locales des processeurs ont chacune un *flot de données* propre. L'addition de deux vecteurs sur une machine à huit processeurs est faite par une opération simultanée sur les huit premières composantes des vecteurs. L'opération *addition* est la même dans tous les processeurs, mais chacun travail sur un élément particulier. Les processeur peuvent communiquer entre eux à travers un réseau. L'avantage de ce genre de machine est

qu'elles sont relativement faciles à programmer. Les langages n'ont pas besoin de changer, car on peut continuer de programmer avec un contrôle séquentiel. La parallélisation est prise en charge par le compilateur (*parallélisme caché*),

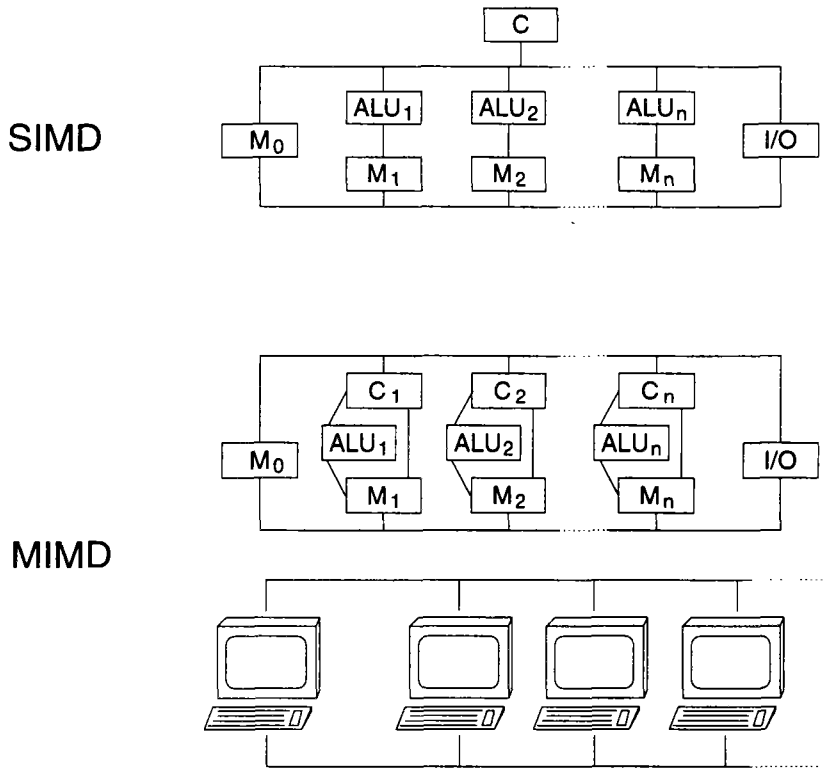


FIG. 4.4 - *L'architectures SIMD n'a qu'un seul contrôle C sur le programme pour tous les processeurs avec l'unité de calcul ALU et la mémoire locale. Généralement on trouve une mémoire qui est accessible pour tous M_0 et des entrées / sorties centralisées I/O. Dans l'architecture MIMD, chacun des processeurs a son propre contrôle sur le programme à exécuter. Les machines inter-connectées en réseau n'ont même pas une mémoire ou des entrées/sorties communes.*

Lorsque l'on multiplie les flot de données *et* les flots de d'instructions on parle de machine MIMD (*multiple instruction stream, multiple data stream*). L'idée d'une machine MIMD est de prendre plusieurs machines et de les faire travailler ensemble. Plusieurs processeurs travaillent en même temps d'une manière indépendante sur des données différentes. On distingue deux façons de partager les données entre les processeurs. Pour une mémoire commune on parle de machine *à mémoire partagée*. Lorsque chaque processeur possède une mémoire qui lui est propre on parle de machine *à mémoire distribuée*. Cette dernière est un ensemble de machines indépendantes exécutant leurs propres programmes sur leur propres données. Il est alors nécessaire d'échanger les informations entre les machines par l'intermédiaire d'un réseau d'interconnexions. Ce réseau peut être une famille de stations de travail reliées par un réseau Ethernet. Suite aux échanges importants de données, on a besoins de liaisons très efficaces entre les machines. Le développement des *transputers* a été spécialement étudié pour ce problème. Ce sont des processeurs avec quatre lignes dédiées pour la communi-

cation. Comme chaque processeur exécute son propre code, il est plus difficile d'écrire des programmes, car il en faut un pour chaque processeur d'une part, et des commandes de communication entre les processeurs d'autre part.

L'application du parallélisme dans le contexte Motor-2

La notion de module dans Motor-2 se prête particulièrement bien à l'extension vers le parallélisme. Il paraît naturel d'associer à chaque module sa propre file d'exécution. L'objet technique a donc son propre objet informatique et sa propre évolution (plus ou moins) indépendante. Une file d'exécution est un flot de contrôle à l'intérieur d'un processus. Un processus peut comprendre plusieurs files d'exécution dont chacune peut être créée et terminée dynamiquement. Elles peuvent partager quelques données globales, mais elles gardent leurs propres variables locales et leur propre avancement du programme.

Nous utilisons donc une machine (virtuelle) de type MIMD à mémoire distribuée. C'est l'architecture qui est la plus adéquate pour représenter des objets indépendants qui ont chacun leurs propres mémoires et leurs propres évolutions. L'évolution de l'ensemble est déterminée par un échange de messages entre les objets.

Faute de disposer d'une vraie machine multiprocesseurs, il faut la simuler. Nous pouvons réaliser cette machine parallèle virtuelle par deux moyens :

- Le langage de programmation Ada contient déjà toutes les constructions nécessaires pour une programmation explicite du parallélisme. Si la machine sur laquelle le programme créé doit tourner n'a pas plusieurs processeurs, le compilateur génère automatiquement les instructions nécessaires pour simuler cette architecture sur une machine monoprocesseur. Le programmeur peut donc construire son programme comme s'il était destiné à un ordinateur multiprocesseur. Le programme se comporte sur une machine monoprocesseur exactement comme sur une machine multiprocesseur[6]. Bien entendu, il sera beaucoup plus lent, mais l'enchaînement des files d'exécution reste garantie.

  Notons que Ada permet un modèle de programmation adapté aux machines à mémoire partagée, ce qui est souvent plus efficace. Malgré cette possibilité, nous nous sommes restreint dans son usage à un échange d'information qui est basé sur l'envoi et la réception explicite des messages. Ceci permet l'application d'un autre outil pour le simulateur Motor-2.

- Une autre façon d'obtenir une machine multiprocesseur est de faire travailler ensemble un réseau de machines monoprocesseurs. Ceci devient facile en utilisant le paquetage PVM. PVM livre des routines qui permettent de connecter et échanger des données entre les machines du réseau. Il est en même temps un moyen pour valoriser le réseau, car il permet une meilleure exploitation des ressources.

6. C'est d'ailleurs l'application typique pour laquelle Ada a été conçu : offrir la simultanéité virtuelle sur des systèmes encapsulés (*embedded systems*).

L'usage mixte des deux approches apporte quelques avantages. Nous pouvons développer le programme dans un environnement sûr en utilisant le langage Ada. On peut se reposer sur le parallélisme prédisposé dans le langage. Une fois le programme vérifié, nous pouvons exploiter le vrai parallélisme des machines inter-connectées par PVM sans trop de difficulté. Le gain en temps, à la fois de développement et de calcul, est certain.

Dans les deux chapitres suivants nous retrouvons l'application et l'implémentation concrète du parallélisme dans le logiciel de simulation Motor-2. Ses influences sur le choix d'algorithme sont traitées au moment où on parle de la résolution numérique.

Chapitre 5

Modules et Raccordement : les deux entités majeures de Motor-2

Nous avons vu dans les chapitres précédents les approches de l'environnement SYMBOL et plus particulièrement Motor-2 pour diminuer la complexité apparente pour l'utilisateur. Le *module* est l'entité de base que l'on obtient par le découpage. Chaque composant du système est représenté par un module dans la description du problème que l'on veut simuler et qui est nécessaire pour reconstruire le système original.

La deuxième notion nécessaire pour la reconstruction du système est le *raccordement*. Il décrit les liens et les dépendances entre les modules pour conserver la cohérence de l'ensemble.

5.1 La notion de module

Nous rappelons brièvement notre terminologie en ce qui concerne la description d'un système. Un *module* est une unité « facilement identifiable »[1] à l'intérieur d'un ensemble ; plus particulièrement, un module est un composant de notre système. La combinaison ou juxtaposition ou mieux le raccordement des modules permet la construction d'un système composé.

À chaque étape du découpage d'un système (voir 4.2), nous appelons les composants obtenus des *modules*. Cela crée deux types de modules. Nous distinguons des *modules élémentaires* qui sont les modules terminaux du découpage (*les feuilles*) et des *modules composés* qui sont des modules de raccordement (*les ramifications*). Un module composé est un ensemble de plusieurs sous-modules et de liens de raccordement qui gèrent les informations sur la connexion.

1. par *facilement identifiable* nous indiquons que l'unité peut être facilement repérée et délimitée dans son environnement. Cette identification n'est pas toujours « facile », et n'est pas une démarche absolue.

Si nous revenons aux deux phases d'une étude de système qui sont la modélisation et la simulation, nous employons le terme *modèle* dans la première phase (construction de modèles) et le terme *module* dans la deuxième phase (un système est composé de plusieurs modules). Lors de la description de notre système réel, nous associons à chaque module un certain type, un *modèle de représentation*. Ceci s'applique surtout pour les modules élémentaires. On dit par exemple que le module du MUR NORD est représenté par un modèle de DIFFÉRENCES FINIES 1D. Pour les modules composés qui sont situés aux niveaux supérieurs de l'arbre de découpage, nous disons tout simplement que le module est du type COMPOSÉ.

Nous avons appliqué aux modules les règles qui traduisent une visibilité limitée. Les deux points de vue d'un objet (voir 4.3) s'appliquent également au module. La vue extérieure spécifie la partie visible d'un module, la vue intérieure contient son implémentation. C'est la vue intérieure qui établie le lien entre un module et son type, le modèle associé. Il doit y avoir aussi cohérence entre les spécifications, la partie visible, d'un module et les entrées/sorties des spécifications du modèle associé. C'est à dire, que l'on ne peut pas associer un modèle qui nécessite trois entrées à un module qui n'a que deux entrées, par exemple.

5.1.1 Vue extérieure

Comme les modules doivent échanger des informations, on est alors conduit à définir des *frontières* de communication. Dans le cas d'un mur par exemple on peut définir les faces gauche et droite comme frontières ; le champ de température à l'intérieur du mur n'est pas vu de l'extérieur. Toute communication avec le module MUR passe par ces frontières GAUCHE et DROITE. Son état interne n'est pas directement visible.

En général, une frontière regroupe plusieurs variables ou ce que nous appelons des *pattes* (*pins*). Ces pattes sont orientées, c'est à dire que les variables sont désignées comme variables d'entrée ou de sortie pour le module. Dans une représentation graphique, nous utilisons des ellipses qui signifient des frontières pour en souligner la nature composée. De plus, nous associons un type à chaque frontière. La face gauche du mur comprend une température et un flux de chaleur. Cette frontière peut être du type DIRICHLET, par exemple. Le type d'une frontière nous donne la possibilité de vérifier la description des connexions. Nous pouvons par exemple refuser le raccordement d'une frontière de type DIRICHLET avec une frontière du type CONVECTION VERTICALE.

L'avantage de la notion d'une *frontière* est de pouvoir remonter à un niveau plus abstrait pour définir complètement la partie visible d'un module. Nous pouvons ainsi rester aux niveaux technique / physique dans la phase descriptive du système, car le raccordement entre modules peut s'exprimer maintenant en termes techniques, plutôt que par une expression mathématique. On dit que la face extérieure du béton est en contact parfait avec la face intérieure de l'isolant. L'interprétation mathématique et algorithmique de cette phrase est connue dans l'environnement Motor-2 pour la simulation.

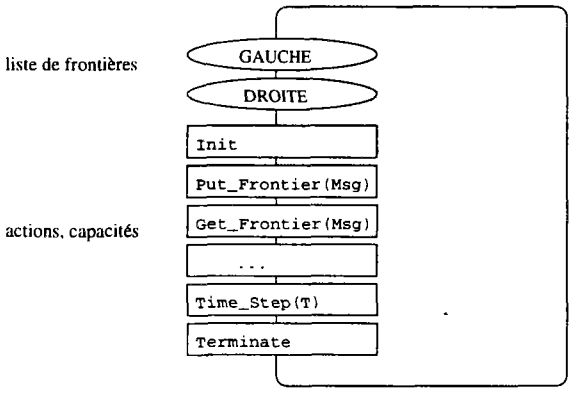


FIG. 5.1 - *Les parties visibles d'un module : les frontières (en haut) spécifient les variables de communication, et les actions (en bas) que le module peut effectuer.*

Dans le cadre de la simulation par Motor-2, les spécifications d'un module sont plus étendues. En utilisant la correspondance entre un composant du système et une file d'exécution séparée (*tâche*), nous ajoutons une qualité supplémentaire au module. Parfois on emploie le terme *acteur* pour désigner cette entité, parce que l'objet créé est une unité de calcul active, une sorte de programme indépendant. Nous avons donc dans la vue extérieure non seulement une partie statique des données (les frontières), mais aussi une partie dynamique. Il faut donc définir les actions qu'un module peut exécuter.

Tout type de module est dérivé d'un type de base abstrait (`Any_Module`), qui définit les actions nécessaires qu'un module doit être capable d'effectuer. Celle-ci sont par exemple l'initialisation et la terminaison de l'objet. Mais le développeur d'un modèle peut ajouter d'autres actions. Une extension à spécifier séparément est éventuellement la capacité du module à calculer des dérivées. Un utilisateur n'a pas besoin de s'occuper des capacités des *modules*, puisqu'elles sont liées au modèle utilisé pour le module. C'est le développeur de *modèle* qui définit les capacités. Tous les modules qui sont de ce type peuvent alors les exploiter.

Vu de l'extérieur, il n'y a pas de différence entre des modules composés et des modules terminaux. L'algorithme de résolution traite tous les modules de la même manière. Ceci est un principe de conception qui permet à Motor-2 de rester un environnement ouvert pour traiter des modèles *a priori* quelconques.

La vue extérieure d'un module étant définie, nous avons le choix entre plusieurs représentations internes. Nous pouvons facilement échanger les modèles associés sans besoin de modifier l'algorithme de raccordement. De cette manière, un modèle plus détaillé peut être exploité pour une partie du système, ou bien un modèle plus simple pour des composants jugés sans importance. L'utilisateur n'est pas obligé de modifier la structure du système. La description du système reste la même. On ne change que le modèle employé par un des composants.

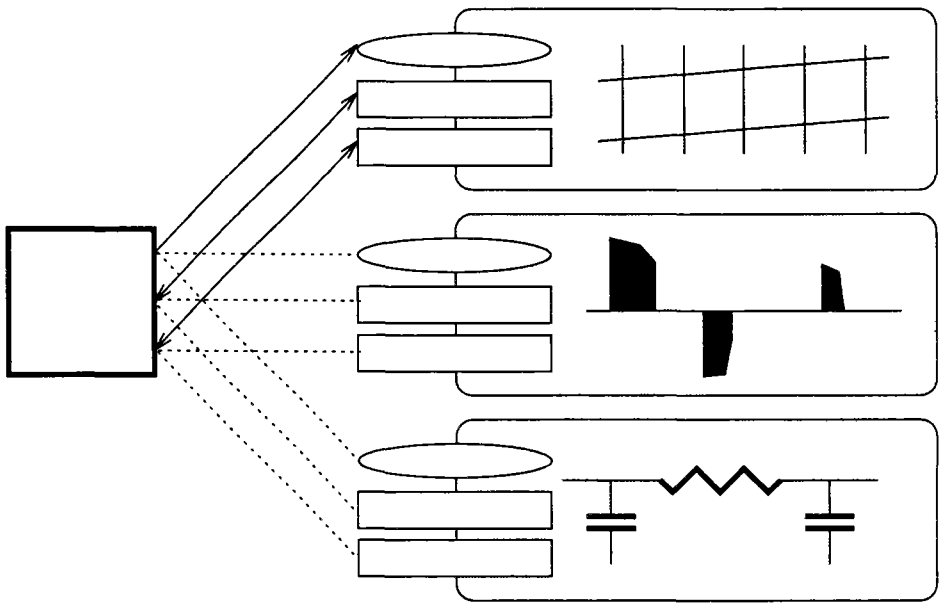


FIG. 5.2 - *Pour une vue extérieure équivalente, on peut échanger les représentations internes des modules.*

5.1.2 Vue intérieure

La partie intérieure d'un module est définie par le modèle qui est associé à celui-ci. L'environnement Motor-2 traduit les appels qui arrivent aux frontières d'un module par des appels correspondants aux fonctions définies par le modèle. Si par exemple le module MUR est de type DIFFÉRENCES FINIES 1D, et reçoit des nouvelles entrées aux frontières, ces valeurs sont automatiquement attribuées aux nœuds extrêmes, et un nouvel état peut être calculé à l'intérieur.

Les modules terminaux se voient attachés leurs propres méthodes locales (contenues dans le modèle). Nous ne nous en occupons pas pour l'instant. Le développement des modèles de calcul est la tâche d'un modélisateur. Ce sujet est traité plus en détail dans le chapitre 7.3.1.

La résolution du système consiste à déterminer les évolutions d'état des modules terminaux ainsi que la résolution locale du couplage. C'est la résolution des raccordements entre les modules. Le raccordement est situé à l'intérieur des modules composés. La vue interne d'un module composé contient donc les vues externes des sous-modules et les contraintes de raccordement. Pour exprimer ces raccordements dans le projet SYMBOL nous utilisons la notion d'*interface*. Le *type* d'une interface décrit la nature du raccordement. Une interface du type CONTACT PARFAIT vérifie l'égalité des températures et la compatibilité des flux de chaleur. Au contraire de ce qui est en usage dans le langage courant, il faut remarquer ici que nous utilisons le terme *interface* pour la communication entre modules et non pour la spécification d'un module qui est une *frontière*.

Aussi les modules composés peuvent être des sous-modules et doivent communiquer avec l'extérieur à travers des frontières. Qu'est ce qu'est une frontière

pour un module composé? Dans un cas simple et général, on peut déclarer une frontière d'un sous-module comme équivalent à la frontière du module composé dans lequel il s'intègre (voir fig. 5.3). Nous extrayions la frontière d'un module fils, et nous la déclarons visible au niveau du père. La face intérieure du BÉTON est en même temps la face intérieure du MUR. Néanmoins il est quelque fois souhaitable de présenter plusieurs frontières internes comme une seule frontière externe. Le mur pourrait avoir deux parties, en béton en bas et en bois en haut par exemple. L'ensemble des frontières intérieures du BOIS et du BÉTON constituent maintenant la frontière INTÉRIEURE du MUR. Dans ce cas, il se pose la question pour l'utilisateur de savoir, quelles variables choisir pour la vue extérieure.

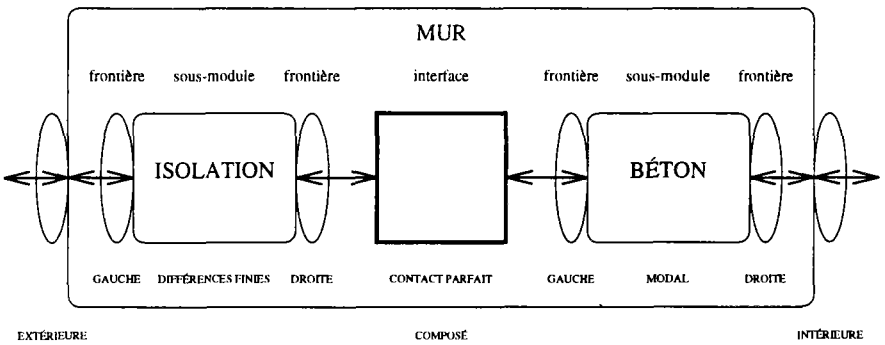


FIG. 5.3 - *Le raccordement entre deux modules: la frontière* DROITE *du module* ISOLATION *est raccordée à la frontières* GAUCHE *du* BÉTON. *Entre les deux se trouve une interface du type* CONTACT PARFAIT. *Pour la communication au prochain niveau supérieur, la frontière* DROITE *du* BÉTON *constitue la frontière* INTÉRIEURE *du module composé* MUR *et de la même manière* GAUCHE *de l'*ISOLATION *constitue la frontière* EXTÉRIEURE *du* MUR.

5.2 Moyens de communication des modules: frontières et pattes

Regardons un module au niveau mathématique plus en détail. Dans ce qui suit le nom d'un module sera indiqué par une lettre latine majuscule. Comme indice, il est mis dans le coin supérieur droit d'une variable.

L'état abstrait d'un module A est représenté par le vecteur η^A. Nous essayons de ne pas utiliser l'état d'un module, car l'état fait partie de la vue interne et nous ne voulons pas dépendre des représentations internes dans nos algorithmes. La communication du module avec l'extérieur passe à travers des *pins* (pattes). Les pattes sont structurées et regroupées en frontières[2]. Une frontière Φ est signifiée par le vecteur de ses pattes Φ_i^A plus un *type* et éventuellement des paramètres supplémentaires. Une frontière regroupe un vecteur

2. Plusieurs frontières peuvent être rassemblées dans un *link*. Dans la plupart des cas il n'y a qu'une seule frontière dans un *link* et le *link* est obsolète. C'est une structure optionnelle qui est nécessaire pour traiter les cas de connexion par recouvrement où on raccorde plusieurs frontières d'un module à plusieurs frontières d'un autre module.

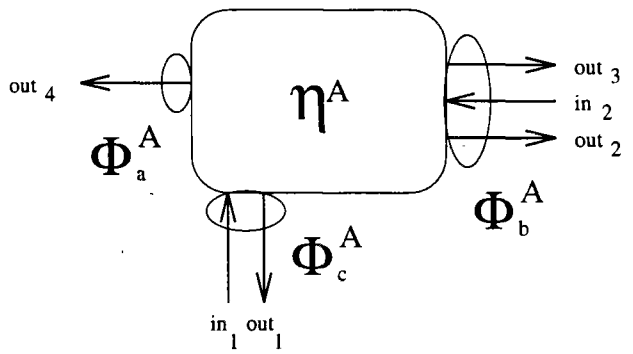


FIG. 5.4 - *Les variables liées au module* A.

local de variables d'entrées $\boldsymbol{e}_i^A$ et un vecteur local de variable de sorties $\boldsymbol{s}_i^A$. Ces variables sont incluses dans le vecteur $\Phi_i^A = \begin{bmatrix} \boldsymbol{e}_i^A \\ \boldsymbol{s}_i^A \end{bmatrix}$. Pour des raisons de cohérence, nous associons toujours un *type* à une frontière. Ce type décrit le nombre et la fonction de ses pattes. Par exemple le type FIRST KIND exige une température en entrée et un flux de chaleur en sortie.

Si l'on ne regarde plus les frontières, mais directement les pattes de connexion, l'ensemble de toutes les pattes d'entrée d'un module sont décrites par le vecteur

$$\boldsymbol{E}^A = \begin{bmatrix} in_1 \\ in_2 \end{bmatrix}, \quad \text{les sorties par le vecteur} \quad \boldsymbol{S}^A = \begin{bmatrix} out_1 \\ out_2 \\ out_3 \\ out_4 \end{bmatrix}. \tag{5.1}$$

On peut également réintégrer la notion de frontière et voir les vecteurs d'entrées et de sorties composés des sous-vecteurs de frontières.

$$\boldsymbol{E}^A = \begin{bmatrix} \boldsymbol{e}_1^A \\ \boldsymbol{e}_2^A \\ \boldsymbol{e}_3^A \end{bmatrix} \quad \text{et} \quad \boldsymbol{S}^A = \begin{bmatrix} \boldsymbol{s}_1^A \\ \boldsymbol{s}_2^A \\ \boldsymbol{s}_3^A \end{bmatrix} \tag{5.2}$$

Parmi les capacités de base d'un module, il y a la possibilité de calculer les sorties en fonction des entrées. Il existe pour un module A une fonction du type

$$\boldsymbol{S}^A = \hat{R}^A(\boldsymbol{\eta}^A, \boldsymbol{E}^A, t) \tag{5.3}$$

Les sorties $\boldsymbol{S}^A$ sont calculées en fonction de toutes les entrées du module $\boldsymbol{E}^A$, de l'état du module $\boldsymbol{\eta}^A$ et du temps t. Souvent on ne regarde qu'une frontière d'un module et on attribue à chacune des frontières d'un module A une fonction R_i^A de la forme

$$\boldsymbol{s}_i^A = R_i^A(\boldsymbol{\eta}^A, \boldsymbol{E}^A, t) \tag{5.4}$$

Nous appelons cette fonction la *réponse* de la frontière Φ_i du module A à l'ensemble des excitations extérieures $\boldsymbol{E}^A$. La réponse de l'équation 5.3 est alors composée de l'ensemble des réponses des frontières de l'équation 5.4.

$$\boldsymbol{S}^A = \begin{bmatrix} \boldsymbol{s}_1^A \\ \boldsymbol{s}_2^A \\ \boldsymbol{s}_3^A \end{bmatrix} = \begin{bmatrix} R_1^A(\boldsymbol{\eta}^A, \boldsymbol{E}^A, t) \\ R_2^A(\boldsymbol{\eta}^A, \boldsymbol{E}^A, t) \\ R_3^A(\boldsymbol{\eta}^A, \boldsymbol{E}^A, t) \end{bmatrix} \tag{5.5}$$

Éventuellement, le module peut savoir calculer localement les sorties d'une frontière en fonction des entrées locales. Si la réponse de la frontière ne prend en compte que les entrées locales nous écrivons

$$\boldsymbol{s}_i^A = \tilde{R}_i^A(\boldsymbol{\eta}^A, \mathbf{e}_i^A, t) \tag{5.6}$$

et on l'appelle la *réponse locale*. Il faut remarquer ici qu'on demande au module de calculer les sorties locales sans connaître toutes les entrées, ou en supposant que les autres entrées n'influencent pas $\boldsymbol{s}_i^A$. En pratique cela signifie que les autres entrées conservent les valeurs qu'elles acquièrent par ailleurs, lorsqu'on observe l'effet de $\boldsymbol{e}_i^A$ sur $\boldsymbol{\eta}^A$ et $\boldsymbol{s}_i^A$. $\tilde{R}$ ne dépend que de $\boldsymbol{e}$, et non de $\boldsymbol{E}$. Cette fonction est comparable avec l'évolution insensible φ_{ins} du FET (voir page 29). Généralement un module ne peut calculer la réponse locale que s'il suppose que les autres entrées gardent leur valeur actuelle ou s'il suppose une évolution donnée. Cette réponse locale est un concept important dans la réalisation de l'algorithme de résolution et également du parallélisme. Nous y revenons plus loin.

Rappelons que nous avons défini trois types de réponses, qu'un module peut être capable de fournir: calculer toutes les sorties en fonction de toutes les entrées ($\boldsymbol{S} = \hat{R}(\boldsymbol{E})$, éq. 5.3), calculer des sorties spécifiques en fonction de toutes les entrées ($\boldsymbol{s} = R(\boldsymbol{E})$, éq. 5.4), et plus particulièrement calculer les sorties locales en fonction des entrées locales ($\mathbf{s} = \tilde{R}(\boldsymbol{e})$, éq. 5.6).

Une frontière sert de lien entre un module et une interface. Sa description complète contient donc

- un *nom*, qui sert d'identificateur,
- ses deux vecteurs de *pattes* pour les entrées et les sorties,
- son *type* pour le contrôle,
- le *module* auquel elle appartient et ...
- l'*interface* à laquelle elle est connectée.

À l'intérieur d'une frontière, on trouve des *pattes*. Cette entité est également décrite par plusieurs attributs:

- un *nom* pour l'identification,

- un *type* (température, débit massique, ...) qui peut être utilisé pour la vérification d'un raccordement,
- une *direction* qui indique la nature de la variable comme sortie ou entrée[3],
- une valeur (généralement un nombre réel),
- une unité physique qui est liée au type de la variable.

Pour améliorer le contrôle de raccordement, et aussi de calcul interne, on pourrait ajouter aussi des valeurs minimale et maximale associées au type de la patte. Cela n'est pas implémenté pour l'instant dans Motor-2.

Comme nous l'avons déjà dit pour le type d'une frontière, le type d'une patte est un moyen pour mieux contrôler la cohérence de la description du système. Nous décrivons par ailleurs ainsi les variables à un niveau plus abstrait (plutôt technique que mathématique) qui permet de les gérer plus facilement. Dans le fichier SYMBOL, on peut ainsi vérifier à première vue le type des variables utilisées.

Ces informations sont utilisées dans le processus de la communication entre les modules. Elle est réalisée dans Motor-2 par les *interfaces* que nous décrivons plus en détail dans les paragraphes suivants.

5.3 Raccordement

5.3.1 Présentation

Comme déjà dit dans les chapitres précédents. nous utilisons une approche dans laquelle on découpe le système et construit un arbre de dépendances. Chaque *embranchement* de l'arbre correspond à des liens entre les modules. Ces liens sont à reconstituer lors de la simulation du système et on les appelle *raccordements*.

Découper un système en sous-systèmes nécessite d'écrire les conditions nécessaires que doivent respecter les composants séparés afin qu'ils représentent le comportement des éléments dans le système. Les comportements des composants sont couplés. Une étape importante est alors la reconstitution du système par *raccordement* des composants. Nous avons défini les *interfaces* comme objets abstraits qui rétablissent les conditions de raccordement.

Le couplage de modules dans un systèmes ne s'exprime pas toujours par des équations de connectique simples. Dès qu'un couplage est autre chose que l'égalité de paires de variables, il y a raccordement et non plus connexion. Les interfaces sont alors des instances de *modèles de couplage*. Un modèle de couplage associé à une interface est par exemple le contact parfait qui vérifie

3. une troisième « direction » *observation* a également été définie comme sortie non-raccordable, mais elle n'est pas utilisée dans Motor-2.

une température égale dans toutes les frontières connectées et une annulation de la somme de flux de chaleur provenant des frontières.

Deux objets-phénomènes différents peuvent être complètement, ou en partie, topologiquement superposés. On n'aura cependant jamais à raccorder deux modules représentant le même phénomène en un même lieu. Si les deux modules concernent le même espace, c'est qu'ils représentent deux phénomènes différents. Cela peut être le rayonnement d'une part et l'échange convectif d'autre part, dans une pièce. Si deux modules sont du même type, les objets sont différents. Par exemple, les couches d'une paroi sont représentées par des modèles de type différences finies.

5.3.2 Topologies de raccordement

Le raccordement peut se réaliser sur des topologies très différentes. Les travaux de BLANC SOMMEREUX ont traité cette problématique pour la construction des *modèles couplés* [8]. Nous le rappelons ici en vue d'une simulation qui exploite des géométries de raccordement différentes. Cela peut éventuellement nous donner la possibilité de ne plus dépendre d'un arbre de découpage, mais plutôt d'un graphe plus général (voir § 4.2).

Si l'on ne regarde que le raccordement de modules par paire, nous pouvons distinguer ces couplages :

- Couplage bord à bord. On raccorde une frontière naturelle de l'un des modèles avec une frontière de l'autre modèle, comme on l'a déjà vu dans l'exemple d'une paroi bicouche.

- Couplage par recouvrement d'une partie commune. Dans la modélisation des bâtiments, on rencontre des situations où un raccordement par recouvrement peut être utile, par exemple pour le couplage de deux zones qui partagent un mur commun. Nous avons mentionné ce problème dans le chapitre « découpage » (§ 4.2). Pour ce type de raccordement les modèles doivent fournir des frontières qui ne se trouvent pas aux extrémités géométriques des composants. Des précaution spéciales sont à prendre, parce que la partie commune est souvent surdéterminée.

- Couplage par recouvrement avec élimination de la partie commune dans l'un des deux modules ou même dans les deux. Ce type de raccordement doit également être prévu pendant la construction du modèle, car il nécessite aussi une frontière interne sur laquelle on peut imposer des sollicitations et il doit fournir toutes les variables de la frontière externe.

En ce moment, seul le couplage bord à bord est implémenté et testé dans le simulateur Motor-2. Les autres topologies de couplage ne sont disponibles que pour la création de *modèles composés*. C'est à dire, avec la description SYMBOL, nous pouvons spécifier la synthèse de *modèles* basés sur des modèles plus élémentaires (*synthèse modale*, voir [8] et [36]).

5.3.3 Styles de description de raccordement

Un des objectifs du projet SYMBOL est de fournir un environnement de travail où on peut exprimer et vérifier le couplage entre les composant au niveau le plus haut, le plus naturel possible. L'idéal serait certainement de pouvoir dessiner deux modules, l'un à côté de l'autre et que le logiciel comprenne qu'il s'agit d'un raccordement de composants.

Dans le § 2.3 nous avons déjà vu quelques méthodes pour décrire le couplage entre les composants d'un système. TRNSYS utilise un fichier où on spécifie les indices des variables de sorties et d'entrée qui sont connectées. Par exemple, la sortie numéro 2 de l'unité 5 est l'entrée numéro 3 dans l'unité 7.

Dans la méthode des Bondgraphs, la description des connexions entre les composant est essentielle. Comme nous l'avons vu, une flèche indique la direction d'un « flux de puissance » qui est échangé entre deux composants. De plus, une marque existe pour spécifier la « causalité » et de cette façon un ordre de calcul préférentiel. Les flux d'information peuvent être ajoutés à ce schéma. Les Bondgraphs donnent une très bonne description des dépendances à l'intérieur d'un système. Leur représentation informatique dans le logiciel TUTSIM passe – similairement à TRNSYS – par des indices. Une connexion correcte entre les composants est vérifiée par le fait que l'ensemble du système d'équations est calculable.

Le format de fichier NMF qui est utilisé dans IDA (et aussi en partie dans Spark) permet non seulement la description de modules élémentaires, mais aussi de modules composés.

Une autre manière de spécifier les dépendances entre les composants d'un système consiste à créer une grande matrice qui a autant de lignes et de colonnes qu'il y a de composants. Les cases de la matrice contiennent les équations qui décrivent la dépendance du module de la colonne en fonction de celui de la ligne.

Dans le cadre du projet SYMBOL nous utilisons des modules composés pour décrire les connexions entre les composants. Le langage utilisé est détaillé au § 7.2. En bref, nous déclarons des interfaces comme points de communication à l'intérieur des modules composés. Ces interfaces ont un type et une liste des frontières qui sont raccordées par cette interface.

5.3.4 Raccordement dans Motor-2 aux différents niveaux d'abstraction

Dans le chapitre 4.1 les différents niveaux d'abstraction ont été présentés. Ces niveaux sont utilisés non seulement pour la description des modules mais aussi pour expliciter les relations entre les modules. Pour aboutir à une structuration pertinente, nous allons suivre l'évolution de la description à l'aide de notre exemple du mur bicouche.

- Dans le *monde technique* le raccordement entre deux objets est une connexion réelle. Une couche de laine de roche est clouée sur un mur en béton.

Dans d'autres cas cela peut être une fenêtre dans un mur ou un tuyau entre un capteur et une pompe.

- Si on passe au *monde physique*, les deux couches sont modélisées par une conduction monodimensionnelle. Pour ce qui concerne la connexion, nous simplifions les singularités des clous (!) et nous négligeons l'inévitable résistance thermique entre les couches. Le raccordement est décrit par le modèle du CONTACT PARFAIT ce qui veut dire qu'il y a une température commune à la surface de contact et le flux de chaleur qui sort du béton entre sans modification dans l'isolation.

- Au *monde mathématique* les modules sont représentés par des équations. En conséquence les relations entre les modules s'expriment soit par des variables communes et des variables de transfert, soit par des équations supplémentaires (prévoir des variables communes une simplification des équations supplémentaires). Pour le contact parfait nous écrivons:

$$T_{l,\text{iso}} = T_{0,\text{béton}} \qquad \text{et} \qquad \phi_{l,\text{iso}} = -\phi_{0,\text{béton}}$$

- Des solutions analytiques existent pour notre exemple si simple. Pour démontrer le cas général nous poursuivons la description au niveau du *monde algorithmique*. Nous utilisons les définition de réponse (éq. 5.4) pour une sorte de raccordement aveugle. Si les deux frontières du raccordement sont du type DIRICHLET, les réponses des modules contiennent le flux de chaleur en fonction de la température imposée. Nous construisons donc une fonction $f = f(R^{\text{ISO}}, R^{\text{BÉT}})$. Dans notre cas, f a la forme

$$f = R^{\text{ISO}}(T) + R^{\text{BÉT}}(T)$$

Nous faisons varier la température jusqu'à ce que la somme des flux de chaleur s'annule. L'algorithme pour trouver la solution est par exemple la méthode de NEWTON-RAPHSON.

5.3.5 Interface

Dans l'environnement Motor-2, nous avons choisi l'*interface* comme lieu (fictif) où s'effectue le couplage entre modules. On y décrit les conditions de raccordement. À l'intérieur d'un module composé on ne spécifie pas seulement le fait qu'il y a des sous-modules, mais on décrit aussi les connexions entre les sous-modules à l'aide des interfaces. Une interface est un point de communication entre des sous-modules fils d'un même module composé parent. C'est donc un regroupement d'un ensemble de connexions vers des frontières des sous-modules (voir fig. 5.3).

Pour pouvoir connecter des frontières très différentes aux interfaces, nous leur associons un *type* qui détermine un *modèle de couplage*. Le type d'une interface spécifie donc la nature du raccordement entre les sous-modules. Pour Motor-2 une interface n'a pas d'épaisseur, c'est à dire qu'elle n'a pas de capacité

physique et qu'il ne se produit aucun phénomène physique. Tout phénomène physique se trouve dans les modules. Comme nous l'avons déjà vu, une interface peut par exemple être du type CONTACT PARFAIT pour un raccordement entre la couche ISOLATION et la couche BÉTON qui constituent ensemble le module composé MUR.

Une interface est un ensemble de relations permettant de « brancher » les parties visibles des modules. C'est l'interface qui sert à *contraindre* les informations des frontières à être compatibles au sens du lien physique entre les modules. Elle même ne porte pas de phénomène physique. Nous regardons le raccordement plus en détail au niveau mathématique et algorithmique – les méthodes de résolution des contraintes – au chapitre 6.

Définition mathématique de l'*interface*

Le raccordement entre des modules est effectuée par des interfaces. Nous indiquons les interfaces par des petites lettres latines et lui associons également un type et des paramètres.

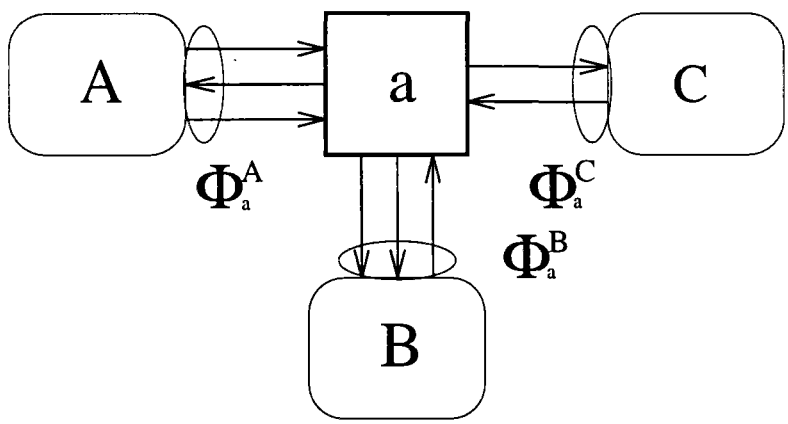


FIG. 5.5 - *Vue de l'interface* a.

Dans la figure 5.5 on voit le raccordement des trois modules A, B et C. Ils sont connectés à l'interface a à travers ses frontières Φ_a^A, Φ_a^B, et Φ_a^C. Cette interface rassemble

- les relations (en général non-linéaires) entre les entrées et les sorties d'une frontière pour chacune des frontières. Ce sont les équations des modules R_a^X.
- une relation entre l'état de l'interface η_a et les variables connectées (les pattes des modules). (L'état provient de l'algorithme de résolution (§ 6.5), il n'a pas de sens physique.)
- des relations contraignant le raccordement et donc l'évolution des modules. Elles sont définies par le *type* de l'interface. Ce type est appelé un *modèle de couplage*. Par exemple le type CONTACT PARFAIT exige égalité

des variables intensives (la température) et compatibilité des variables extensives (flux de chaleur))

Prenons maintenant l'exemple de la conduction monodimensionelle dans un mur bicouche composé d'une couche A et d'une couche B. Les conditions aux limites de deux couches étant de première espèce, elles sont raccordées par l' interface a de type CONTACT PARFAIT. Le type de l'interface détermine les relations de raccordement.

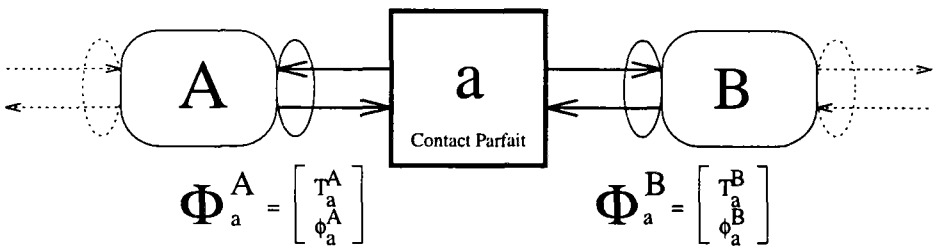


FIG. 5.6 - *Exemple d'un mur bicouche.*

Les contraintes du contact parfait sont une température commune et le bilan nul de flux de chaleur. Cela s'exprime par

$$T^A = T^B \quad \text{et} \quad \phi^A + \phi^B = 0. \tag{5.7}$$

Les frontières des modules A et B vers l'interface a sont composées d'une température et du flux correspondant

$$\Phi_a^A = \begin{bmatrix} T^A \\ \phi^A \end{bmatrix}, \quad \Phi_a^B = \begin{bmatrix} T^B \\ \phi^B \end{bmatrix}. \tag{5.8}$$

Les relations entre les variables d'une frontière sont les équations des modules. Les conditions aux limites étant de première espèce par exemple, on peut exprimer les flux en fonction des températures.

$$R^A(T^A, \eta^A, t) = \phi^A, \quad R^B(T^B, \eta^B, t) = \phi^B. \tag{5.9}$$

On peut rassembler ces équations dans une écriture symbolique de forme

$$\begin{array}{ll} R_a^A(T^A, \eta^A, t) & = \phi^A \\ R_a^B(T^B, \eta^B, t) & = \phi^B \\ \text{température commune} & \\ \text{égalité de flux} & \end{array} \quad \begin{bmatrix} R_a^A & 0 & -1 & 0 \\ 0 & R_a^B & 0 & -1 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} T^A \\ T^B \\ \phi^A \\ \phi^B \end{bmatrix} = \boldsymbol{0} \tag{5.10}$$

R^A et R^B signifient ici les relations non-linéaires entre les températures imposées et les flux de chaleur des modules.

Généralisation

À ce stade il n'y a pas d'intérêt particulier à distinguer entre entrée et sortie d'un module. C'est l'ensemble des variables d'une frontière Φ qui est traité par le module. L'équation (5.6) qui calcule la réponse locale

$$\tilde{R}_i^A(\boldsymbol{\eta}^A, \boldsymbol{e}_i^A, t) - \boldsymbol{s}_i^A = 0 \tag{5.11}$$

devient alors

$$\tilde{f}_i^A(\Phi^A) = 0. \tag{5.12}$$

De la même façon nous pouvons aussi exprimer l'équation 5.4 qui calcule la sortie locale en fonction de toutes les entrées d'un module

$$R_i^A(\boldsymbol{\eta}^A, \boldsymbol{E}_i^A, t) - \boldsymbol{s}_i^A = 0 \tag{5.13}$$

Elle devient

$$f_i^A(\Phi^A) = 0 \tag{5.14}$$

Ici nous ne mentionnons plus la dépendance entre la frontière Φ et l'état $\boldsymbol{\eta}^A$ et le temps t. Ces dépendances sont « cachées » dans le module et ne sont pas « visibles » pour l'interface. L'algorithme n'a pour résoudre le système d'équations de l'interface que les variables présentes dans la frontière.

Un module A calcule $f_a^A(\Phi^A)$ dans sa frontière connectée à l'interface a Les équations de contraintes relient les pattes des modules raccordés à l'intérieur de l'interface. Dans le cas général, les équations des contraintes peuvent être également non-linéaires, et peuvent inclure une dépendance de l'état de l'interface $\boldsymbol{\eta}_a$, mais elles ne sont jamais discontinues[4]. Dans le § 6.5, nous voyons comment l'état d'une interface peut être utilisé dans la simulation. L'ensemble des équations d'une interface est alors un système d'équations non-linéaires (SENL) qu'il faut résoudre. Les algorithmes pour résoudre des SENL cherchent à trouver des valeurs de variables pour lesquelles toutes les fonctions s'annulent.

$$\begin{array}{lrcccccl}
 & f^A(& \Phi_a^A, & 0, & 0, & \cdots, & 0) & = 0 \\
 & f^B(& 0, & \Phi_a^B, & 0, & \cdots, & 0) & = 0 \\
\text{équations des modules} & f^C(& 0, & 0, & \Phi_a^C, & \cdots, & 0) & = 0 \\
 & \vdots & & & & & & \\
 & f_1^a(& \Phi_a^A, & \Phi_a^B, & \Phi_a^C, & \cdots & \boldsymbol{\eta}_a) & = 0 \\
 & \vdots & & & & & & \\
\text{équations des contraintes} & f_n^a(& \Phi_a^A, & \Phi_a^B, & \Phi_a^C, & \cdots & \boldsymbol{\eta}_a) & = 0
\end{array} \tag{5.15}$$

Le système (5.15) a essentiellement deux parties. L'une contient les équations indépendantes R_a des modules. Ici on ne trouve pas de connexion entre

4. Toute éventuelle discontinuité est dans la réponse du module R.

les frontières et leurs variables associées. Comme nous ne regardons que des variables « visibles » pour l'interface, les dépendances des états des modules ne sont pas mentionnées dans cette écriture. On remarque déjà qu'il n'y a des éléments que dans la diagonale principale. L'algorithme de résolution doit prendre en compte ce fait afin d'améliorer l'efficacité numérique. La deuxième partie contient les équations de raccordement qui sont propre à l'interface et qui dépendent de son type. Ces équations relient les frontières entre elles, et éventuellement avec un état de l'interface.

Nous cherchons alors la solution d'un système d'équations non-linéaires (SENL). Pour des connexions purement physiques, nous savons qu'une solution existe dans le monde réel. Dans le monde mathématique ou algorithmique le problème peut être mal posé. Cela peut nous compliquer la tâche ou même empêcher de trouver la solution.

Le chapitre suivant traite les différentes méthodes qui ont été implémentées dans le simulateur Motor-2 pour résoudre les systèmes d'équations qui apparaissent dans un module composé. L'influence d'une parallélisation de l'exécution sur les calculs a une place importante dans ce contexte.

Chapitre 6

Stratégies de résolution des raccordements

6.1 Motor-2 : solveur de raccordements

Après avoir défini la formulation mathématique dans le chapitre précédant, nous passons maintenant au niveau algorithmique et regardons les méthodes implémentées pour la simulation du système.

Les modules composés sont les porteurs principaux de la résolution numérique dans la simulation d'un système avec Motor-2. Les interfaces qui gèrent la communication entre les sous-modules assurent le respect des contraintes de raccordement du système du monde technique. Ces contraintes s'expriment par des systèmes d'équations non-linéaires qu'il faut résoudre dans chacune des interfaces. Le problème principal de Motor-2 pour la simulation se situe donc plutôt dans la résolution des raccordements que dans les calculs des modules terminaux. Son travail consiste à trouver des solutions dans les interfaces. Il s'appui sur la capacité des sous-module à fournir des réponses aux excitations dans les frontières. Les modules terminaux contiennent chacun leur propre code de calcul. Motor-2 ne résout que la structure d'interfaces en faisant appel aux méthodes contenues dans les modules terminaux.

Pour la résolution d'un système d'équations non-linéaires, l'algorithme générique est basé sur la méthode de NEWTON-RAPHSON. C'est une méthode itérative pour s'approcher de la solution en suivant la pente des fonctions. Parmi les numériciens, elle est considérée comme la seule méthode sûre et efficace. Malgré cela un bon nombre de logiciels utilisent la méthode du *point fixe*. C'est certainement dû à une implémentation plus simple. Alternativement on applique deux fonctions sur un jeu de variables ($x = f(y)$ et $y = g(x)$). Les résultats ne sont pas toujours très bons et la convergence n'est pas assurée. Pour ce qui concerne la méthode de NEWTON-RAPHSON, il existe quelques algorithmes modifiés dont les performances sont supérieures à celles de la méthode originale (voir annexe D.1.1). Toutes ces méthodes nécessitent la matrice des dérivées

(souvent appelée la *Jacobienne*). La résolution passe généralement par une inversion de cette matrice. Cette opération peut poser beaucoup de problèmes au niveau numérique : mauvais conditionnement, erreurs d'arrondis, etc. Si la *Jacobienne* doit être déterminée numériquement par perturbation des entrées, les calculs sont encore plus nombreux.

Généralement une simulation essaye de déterminer le comportement d'un système dans le temps. La formulation mathématique standard pour exprimer une dépendance du temps est une équation différentielle. Ce type d'équation doit être intégré dans le temps. Nous supposons que cette intégration se fait à l'intérieur des modules terminaux. Si l'intégration était à faire dans les interfaces, on devrait rendre visible l'état (et la dérivée) des modules dans les frontières. Cela va à l'encontre de notre approche orientée objet et de la visibilité de données restreinte. Nous nous contentons donc d'une solution des systèmes d'équation non-linéaires dans les interfaces.

Pour obtenir une simulation plus efficace, nous avons donc intérêt à diminuer les calculs nécessaires et à exploiter notre structure de description de liens. Dans les algorithmes implémentés nous profitons du fait que le système d'équations est creux – au moins dans sa première partie. Nous faisons la distinction entre des résolutions *globales* et *locales*. Les premières travaillent sur l'ensemble des équations d'un module composé, les dernières traitent les interfaces séparément.

De plus, les composants du système à étudier sont des modules actifs. Les calculs dans chacun des modules sont temporellement indépendants. L'exploitation du parallélisme entre les calculs des modules influence également les algorithmes de résolution.

La hiérarchisation des calculs, qui provient de la hiérarchie obtenue à l'occasion de la description du système est un autre point important qui influence l'efficacité des algorithmes. Ce dernier point ainsi que la pertinence d'une structure de découpage sont traitées dans le chapitre 9.

6.2 Module composé

Pour l'instant, nous nous limitons à regarder le problème de la résolution à un seul niveau, c'est à dire, à l'intérieur d'un seul module composé. Si les sous-modules sont également des modules composés, ils trouvent leur solution de la même manière. Les appels à ces sous-modules sont des excitation extérieures pour ceux-ci et on applique de nouveau le même algorithme pour un module composé. La racine de toute description de système est un module composé pour lequel toutes les entrées sont des constantes (les valeurs initiales des pattes d'entrée). L'arbre du découpage est résolu récursivement.

6.2.1 Méthode générale

À l'intérieur d'un module composé, on trouve un certain nombre de sous-modules et des interfaces. Les interfaces servent comme point de communication

entre les sous-modules.

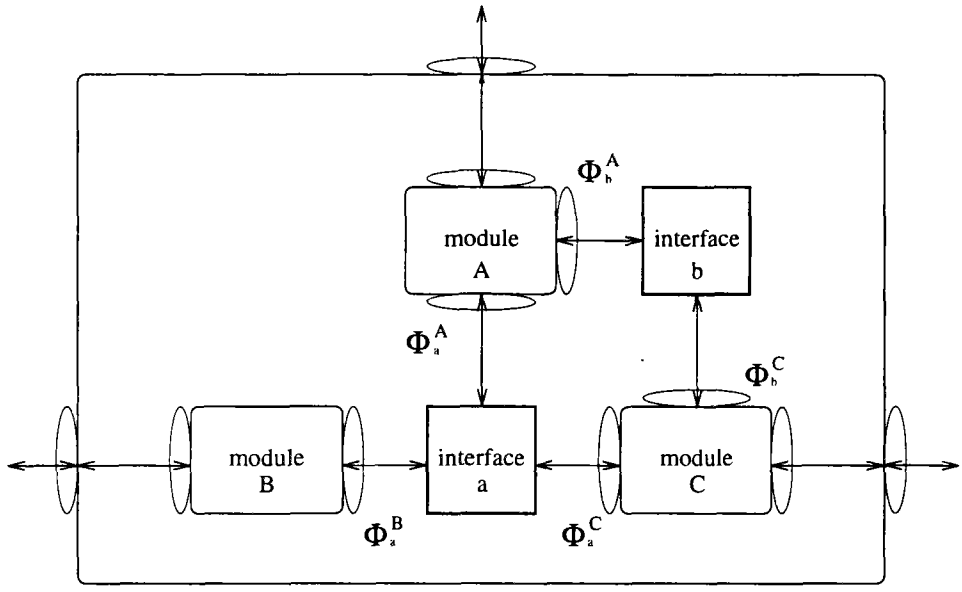


FIG. 6.1 - *Un* module composé *de trois sous-module (A, B, C) et deux interfaces entre ces sous-module (a, b). Trois frontières sont directement liées à des frontières du module père.*

Soit un module composé de S sous-modules et I interfaces entre les sous-modules. Les frontières extérieures du module composé sont également présentes. Elles assurent la communication avec ses modules-« frères » au niveau immédiatement supérieur. Le module composé de la figure 6.1 contient trois sous-modules et deux interfaces.

Dans ce qui suit, on suppose que les valeurs des frontières extérieures sont constantes et ses valeurs sont distribuées aux sous-modules correspondants. La tâche principale d'un module composé est de déterminer les valeurs des interfaces. Avec nos I interfaces, il y a maintenant I systèmes d'équations du type de l'équation 5.15. L'ensemble de ces I systèmes constitue un grand système d'équations à l'intérieur d'un module composé. Nous appelons la matrice qui est constituée par ce système la *matrice globale de raccordement* $\boldsymbol{C}$.

Cette matrice $\boldsymbol{C}$ est structurée par blocs qui se trouvent tous sur (ou sous) la diagonale principale. Schématiquement elle a la structure de la figure 6.2. Chacun des blocs représente le système d'équations d'une interface. Ces blocs ne sont pas carrés à cause des équations propres à l'interface qui relient les sous-modules, les équations de contrainte (voir schéma 5.15). Dans leur partie supérieure, ils n'ont que des éléments sur la diagonale principale. Ceci résulte de notre formulation par frontière qui expriment les réponses de modules aux excitations. Les deux sous-systèmes d'équations pour les interfaces a et b de la figure 6.1 sont représentés dans l'expression 6.1. Nous avons supposé une seule équation de contraintes dans l'interface a (f_1^a). L'interface b contient deux équations de contraintes (f_1^b et f_2^b).

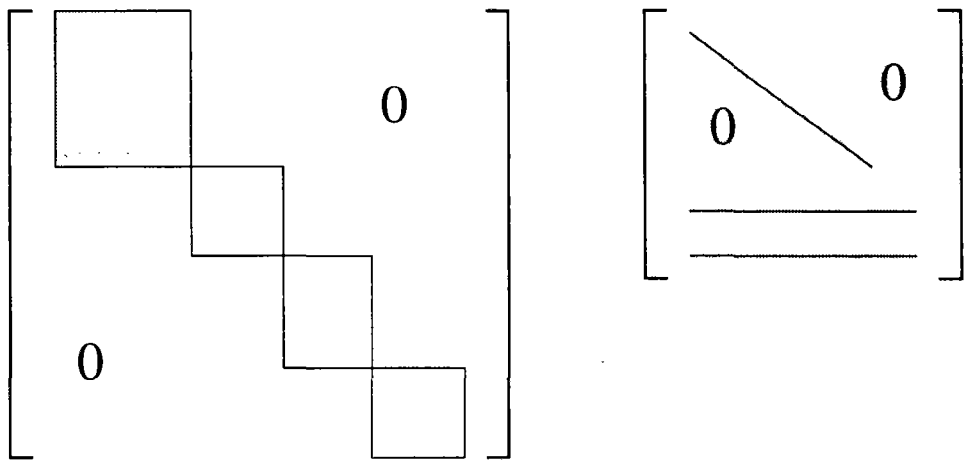


FIG. 6.2 - *La structure de la* matrice globale de raccordement $\boldsymbol{C}$ *d'un module composé (à gauche) et structure d'une sous-matrice (à droite) qui correspond à une interface. Ces matrices ne sont généralement pas carrées.*

$$
\begin{array}{llllllll}
f_a^A(& \Phi_a^A, & 0, & 0, & 0, & 0, & 0, & 0) = 0 \\
f_a^B(& 0, & \Phi_a^B, & 0, & 0, & 0, & 0, & 0) = 0 \\
f_a^C(& 0, & 0, & \Phi_a^C, & 0, & 0, & 0, & 0) = 0 \\
f_1^a(& \Phi_a^A, & \Phi_a^B, & \Phi_a^C, & \eta_a, & 0, & 0, & 0) = 0 \\
f_b^B(& 0, & 0, & 0, & 0, & \Phi_b^A, & 0, & 0) = 0 \\
f_b^C(& 0, & 0, & 0, & 0, & 0, & \Phi_b^C, & 0) = 0 \\
f_1^b(& 0, & 0, & 0, & 0, & \Phi_b^A, & \Phi_b^C, & \eta_b) = 0 \\
f_2^b(& 0, & 0, & 0, & 0, & \Phi_b^A, & \Phi_b^C, & \eta_b) = 0
\end{array}
\tag{6.1}
$$

6.2.2 Réduction des équations

Une optimisation de la formulation consiste à réduire le nombre d'équations qui sont à traiter par interface. La partie supérieure d'une sous-matrice peut bien souvent être supprimée en exprimant adroitement les équations de raccordement. En effet si l'on revient au caractère entrée/sortie des fonctions de frontières ($\Phi = \begin{bmatrix} \boldsymbol{e} \\ \boldsymbol{s} \end{bmatrix}$), et en limitant les types de frontières à certains types permis, on peut utiliser directement les entrées et sorties dans les équations de contraintes.

Les démarche de cette optimisation peuvent être montrées à l'aide d'un exemple. Reprenons celui du chapitre précédant, où nous exprimons les équations d'une interface CONTACT PARFAIT entre deux couches. Les couches ont

des frontières de type PREMIÈRE ESPÈCE. La formulation générale donne

$$\begin{bmatrix} R_a^A & 0 & -1 & 0 \\ 0 & R_a^B & 0 & -1 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} T^A \\ T^B \\ \phi^A \\ \phi^B \end{bmatrix} = \boldsymbol{0} \tag{6.2}$$

pour une température commune et une égalité des flux. Cela fait deux équations pour les frontières et deux équations de contraintes.

Les sorties des modules sont calculées de façon à ce qu'ils vérifient l'équation 5.12 et donc, $R(\boldsymbol{E}) - \boldsymbol{s} = 0$, l'équation de frontière pour une interface est toujours vraie. Il n'est donc pas nécessaire de les inclure dans le système d'équation d'une interface. On utilise plutôt directement les entrées ($\boldsymbol{E}$) et sorties ($\boldsymbol{s}$). Si dans une interface de type CONTACT PARFAIT toutes les frontières sont du type PREMIÈRE ESPÈCE, les fonctions de l'interface peuvent être réduites à une seule. Avec

$$T^A = T^B = T \quad \text{et} \quad R^A = \phi^A, R^B = \phi^B \tag{6.3}$$

on obtient finalement

$$f(T) = \phi^A(T) + \phi^B(T) = 0. \tag{6.4}$$

C'est cette seule équation ($f(T) = 0$) qui est soumise à l'algorithme de résolution. Bien entendu, on perd en généralité, mais on gagne en efficacité pour les connexions connues (une seule équation au lieu de quatre, mais toutes les frontières doivent être du même type PREMIÈRE ESPÈCE). De cette façon un nouveau type d'interface a été créé afin d'appliquer cette optimisation. C'est à l'utilisateur de spécifier s'il veut qu'elle soit appliquée pour un raccordement donné. Les algorithmes de raccordements disponibles pour une interface sont stockés dans une bibliothèque de modèles d'interface. Pour la description détaillée de la façon dont l'utilisateur peut créer ses propres modèles de raccordement (les types d'interface), voir § B.2.

6.2.3 Dépendances cachées

Dans l'écriture généralisée (éq. 5.12) n'apparaît pas la dépendance entre la frontière et l'état interne, ce qui fait que nous ne voyons plus explicitement les dépendances entre les interfaces dans la matrice globale de raccordement. Tous les sous-systèmes semblent indépendants. Mais puisque les mêmes modules sont concernés par plusieurs interfaces, les sous-systèmes sont connectés, bien que la dépendance ne soit pas directement apparente. Dans notre exemple de la figure 6.1, on note que Φ_b^A (la frontière du module A qui est connectée à l'interface b) et Φ_a^A dépendent tous les deux de l'état $\boldsymbol{\eta}^A$ du module A, les réponses dans les frontières Φ_a^A et Φ_b^A ne sont donc pas indépendantes. De la même manière Φ_a^C et Φ_b^C dépendent de l'état du module C. Les interfaces a et b (et leurs systèmes d'équations) s'influencent alors à travers les dépendances des états internes des modules.

6.3 Différentes stratégies de résolution

Des approches différentes ont été suivies pour résoudre le système d'équations internes d'un module composé. Elles se distinguent par deux aspects ; premièrement une résolution globale (par module composé) par rapport à une résolution locale (par interface), et deuxièmement un enchaînement d'exécution synchrone par rapport à une exécution asynchrone entre les sous-modules. Les approches sont présentées dans les paragraphes suivants.

- La première approche résout le système global d'un module composé (la matrice $\boldsymbol{C}$ comme dans 6.1 comme *un* grand système d'équations (la méthode GLOBALE).

- Comme le schéma de la figure 6.2 le montre, les systèmes des interfaces semblent indépendants. Une deuxième approche traite les sous-systèmes indépendamment les uns des autres (la méthode LOCALE). Ceci évite le traitement des grandes parties vides de la matrice $\boldsymbol{C}$ (fig. 6.2).

- L'idée d'indépendance entre les interfaces peut être approfondie. La troisième approche est basée non seulement sur l'indépendance entre les interfaces mais aussi entre les frontières du même module. Sa réalisation exploite encore plus les possibilités du parallélisme (méthode ASYNC). L'interaction des interfaces avec les sous-modules n'est plus synchronisée. La réponse obtenue sur une frontière ne prend pas forcément en compte l'évolution sur les autres frontières du même module.

- Une quatrième méthode renonce aux itérations pour déterminer les interfaces. Les valeurs de sortie du premier modules sont directement distribuées aux entrées des autres modules (méthode DIRECT). Cela ne fonctionne qu'à la condition que l'ordre de calcul est bien spécifié par l'utilisateur et les interfaces sont assez simples pour le permettre.

Les termes « global » et « local » sont utilisés ici dans un sens particulier. En fait, « global » exprime la vue globale seulement à l'intérieur d'un module composé. On regarde ici toutes les interfaces à la fois par rapport à la vue locale où chacune des connexions est traitée séparément.

6.3.1 La méthode globale

Rappelons les unités que l'on trouve dans un module composé. Il y a premièrement les sous-modules avec ses frontières. À chacune des frontières est associée une fonction définie comme dans le chapitre précédent (éq. 5.12) qui remet à jour toutes les composantes de la frontière $\Phi = f(\Phi)$ en mettant à jour l'état interne. Viennent ensuite les interfaces pour la communication entre les modules. Les contraintes du raccordement sont exprimées par des équations associées aux interfaces. Comme troisième point, on trouve les frontières propres

du module composé même. Enfin, nous avons les variables d'entrée et de sortie. Les entrées sont considérées constantes dans ce qui suit, les sorties sont à déterminer.

Toutes les variables et fonctions qui apparaissent dans les frontières et les interfaces sont rassemblées en un seul système d'équations, la matrice de raccordement $\boldsymbol{C}$. L'algorithme de résolution de la méthode GLOBALE traite l'ensemble de toutes ces fonctions. La matrice globale $\boldsymbol{C}$ qui représente le système d'équations est traitée comme une matrice pleine. Nous ne prenons aucune mesure particulière pour ce qui concerne sa faible densité.

Comme le système est complet, il n'est pas nécessaire de se restreindre aux réponses locales dans les frontières. Les modules peuvent calculer les sorties s'ils connaissent toutes les entrées. Nous utilisons donc des équations du type de 5.3 ($\boldsymbol{s} = R(\boldsymbol{E})$ $\Phi = f(\Phi)$) dans les frontières. Il n'y a pas d'indépendance à respecter entre les interfaces. Les sorties peuvent être calculées en fonction de toutes les entrées.

Les dépendances entre les variables des interfaces ne sont pas directement visibles dans la structure du système comme dessinée dans la figure 6.2. Elles sont dues aux influences mutuelles des frontières d'un même module. Le traitement de la matrice $\boldsymbol{C}$ comme une matrice pleine fait apparaître ces dépendances pendant la résolution. La matrice *Jacobienne* des dérivées entre toutes les variables n'a plus la même structure que la matrice $\boldsymbol{C}$ de la figure 6.2. Elle a des éléments non-nuls aussi hors de la diagonale principale. Ces éléments hors diagonale sont des influences certaines, mais non spécifiées par l'utilisateur dans la description du système.

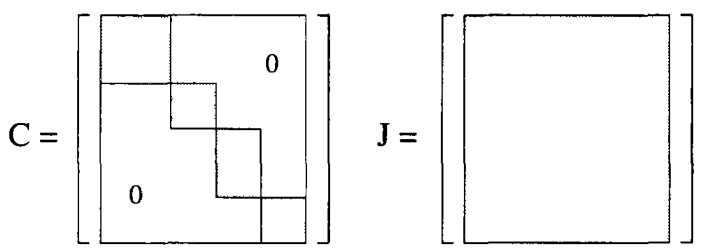


FIG. 6.3 - *La méthode* GLOBALE *pour un module composé traite la matrice* $\boldsymbol{C}$ *comme une matrice pleine (à gauche). Les sous-modules calculent les sorties en fonction de toutes les entrées. Dans la matrice des dérivées, nous trouvons des éléments non-nuls partout (à droite).*

La méthode GLOBAL prend en compte toutes les variables et toutes les fonctions existantes dans un modules composé. L'algorithme de NEWTON-RAPHSON avec lequel est traité le système d'équations applique des corrections sur toutes les variables en même temps. C'est la méthode la plus sûre, elle est à utiliser surtout en cas de problème numérique comme par exemple celui qui provient des discontinuités dans l'évolution des sous-modules.

Si on sait *a priori* que les éléments de la *Jacobienne* hors diagonale principale sont petits par rapport aux autres, on peut s'épargner les efforts de calculs sur la matrice $\boldsymbol{C}$ complète en utilisant la méthode de traitement LOCAL.

6.3.2 La méthode locale

Un deuxième algorithme pour résoudre un module composé consiste à traiter les interfaces séparément les unes des autres. Ceci veut dire que l'on coupe la matrice $\boldsymbol{C}$ (fig. 6.2) en autant de sous-matrices qu'il y a d'interfaces dans le module composé. Chaque système d'équations résultant est un problème entier qui est résolu séparément des autres. Les équations des frontières sont également du type de l'équation 5.4 ($\boldsymbol{s} = \boldsymbol{R}(\boldsymbol{E})$ $\Phi = f(\Phi)$). Nous ne considérons que les sorties locales. Mais elles sont toujours calculées en fonction de toutes les entrées d'un module. Quand les calculs pour toutes les interfaces sont terminés, toutes les entrées des sous-modules sont donc connues, et on lance les calculs des sous-modules pour qu'ils répondent avec les nouvelles valeurs de sorties.

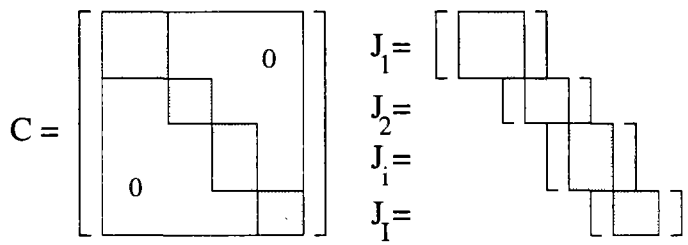


FIG. 6.4 - *La méthode* LOCALE *pour un module composé traite la matrice* $\boldsymbol{C}$ *comme* I *systèmes d'équations indépendants. Les sous-modules calculent toujours les sorties en fonction de toutes les entrées. Nous avons donc également* I *matrices de dérivées (à droite).*

Nous pouvons espérer obtenir des résultats corrects seulement si les influences entre les interfaces ne sont pas importantes. La résolution n'utilise que des informations localement présentes. L'idée est que chaque module « négocie » avec son environnement l'état aux frontières. Nous négligeons donc les dépendances de l'interface qui proviennent du système global. Mais si ces influences sont grandes, nous commettons des erreurs importantes à chaque pas d'itération. Cela peut nous empêcher de converger et de trouver des solutions cohérentes entre les interfaces. Il faut donc que les influences entre les interfaces soient les plus faibles possibles.

Par rapport à l'approche précédente, cette méthode LOCALE évite déjà l'évaluation des grandes parties vides de la matrice $\boldsymbol{C}$ (fig. 6.2). Ceci diminue considérablement le travail du programme. D'un autre côté, des itérations supplémentaires peuvent éventuellement apparaître à cause des éléments qui sont maintenant négligés dans la matrice *Jacobienne*.

Les dépendances entre les interfaces ne deviennent « sensibles » que par appels aux modules. L'information d'un changement dans une des interfaces doit d'abord traverser un module pour être sensible dans une autre interface. Cela fait retarder le flux d'information entre les interfaces, si le module ne la transmet que partiellement. Dit d'une autre façon, la solution locale d'une interface doit dépendre davantage des réponses locales aux changements locaux

qu'aux changements plus distants. Mathématiquement, il faut vérifier que

$$\left|\frac{\partial s_i}{\partial e_i}\right| >> \left|\frac{\partial s_i}{\partial e_j}\right|_{i \neq j}$$

La variation d'une sortie doit être plus influencée par des entrées qui sont dans la même frontière que par des entrées des autres frontières. Ceci est le cas si le module est très inerte. Ce n'est pas le cas pour les modules qui calculent directement l'état stationnaire, ou pour le rayonnement par exemple. Ici les sorties dépendent également de toutes les sorties. Mais on peut facilement construire des modèles où les sensibilités sont inversées, c'est à dire la valeur d'une sortie dépend plus d'une autre entrée que de l'entrée qui est dans la même frontière (régulateurs, par exemple). Les problèmes thermiques typiques contiennent presque tous, sauf exception (!), des capacités.

Exemple : sensibilité locale des sorties sur les entrées Nous allons démontrer sur un exemple que les influences des entrées sur les sorties d'une même frontière sont plus importantes que les influences des autres entrées sur les sortie de cette frontière, lorsque la définition de celle-ci est dictée par des considérations topologiques. Cela peut être justifié si les modules ont des capacités internes.

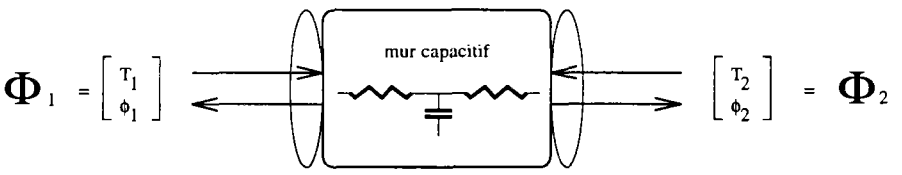


FIG. 6.5 - *Un mur mince avec capacité en conduction monodimensionelle.*

Prenons l'exemple d'une capacité avec deux frontières comme une couche mince d'un mur, dans laquelle on modélise la conduction monodimensionelle (fig. 6.5). Cette couche peut être représentée par une capacité sollicitée par des températures des deux côtés :

$$C\dot{T}_c = h_1(T_1 - T_c) + h_2(T_2 - T_c) \tag{6.5}$$

Ce module a deux frontières Φ_1 et Φ_2 de première espèce, composées de température T_i et flux de chaleur $\phi_i = h_i(T_i - T_c)$: $\Phi_1 = \begin{bmatrix} T_1 \\ \phi_1 \end{bmatrix}$ et $\Phi_2 = \begin{bmatrix} T_2 \\ \phi_2 \end{bmatrix}$. Nous allons estimer l'influence des deux températures T_1 et T_2 sur une des réponses, le flux ϕ_1. C'est la sensibilté d'une sortie aux entrées différentes, donc $\frac{\partial \phi_1}{\partial T_1}$ et $\frac{\partial \phi_1}{\partial T_2}$ respectivement.

La transformation de *Laplace* de l'équation 6.5 donne

$$C[\mathcal{L}(T_c)s - T_c(0)] = h_1[\mathcal{L}(T_1) - \mathcal{L}(T_c)] + h_2[\mathcal{L}(T_2) - \mathcal{L}(T_c)]$$

et on obtient

$$\mathcal{L}(T_c) = \frac{h_1\mathcal{L}(T_1) + h_2\mathcal{L}(T_2)}{Cs + h_1 + h_2} + \frac{T_c(0)}{s + \frac{h_1+h_2}{C}} \tag{6.6}$$

Les deux entrées T_1 et T_2 considérées comme constantes (ou, ce qui revient au même, un échelon à $t = 0$) donne la forme

$$\mathcal{L}(T_c) = \frac{h_1T_1/s + h_2T_2/s}{Cs + h_1 + h_2} + \frac{T_c(0)}{s + \frac{h_1+h_2}{C}}$$

et après une retransformation et une simplification, la solution pour T_c s'écrit :

$$T_c = \frac{h_1T_1 + h_2T_2}{h_1 + h_2} - \frac{e^{-\frac{h_1+h_2}{C}t}}{h_1 + h_2}(h_1T_1 + h_2T_2) - T_c(0)e^{-\frac{h_1+h_2}{C}t} \tag{6.7}$$

Nous pouvons alors déterminer le flux $\phi_1 = h_1(T_1 - T_c)$ et sa sensibilité sur une variations des entrées T_1 et T_2.

$$\frac{\partial\phi_1}{\partial T_1} = h_1 - \frac{h_1^2}{h_1 + h_2}\left(1 - e^{-\frac{h_1+h_2}{C}t}\right) \tag{6.8}$$

$$\frac{\partial\phi_1}{\partial T_2} = -\frac{h_1h_2}{h_1 + h_2}\left(1 - e^{-\frac{h_1+h_2}{C}t}\right) \tag{6.9}$$

On voit ici qu'au début de l'évolution (quand t est encore proche de 0), la réponse est totalement déterminée par l'excitation locale de la frontière 1. L'excitation de la frontière 2 n'est pas encore visible. Nous obtenons

$$\left.\frac{\partial\phi_1}{\partial T_1}\right|_{t=0} = h_1 \qquad \left.\frac{\partial\phi_1}{\partial T_2}\right|_{t=0} = 0$$

Si t tend vers ∞, les excitations se sont égalisées. Leur influence sur le flux ϕ_1 ne dépend que des valeurs des résistances. Elle a pour les deux termes le même ordre de grandeur.

Pour conclure, on peut dire que plus la capacité est grande, plus l'influence des autres frontières sur la sortie locale à travers le module est retardée. Tout au début de l'évolution elle n'existe même pas. La visibilité des autres entrées sur la sortie locale dépend alors du pas de temps. Un pas de temps court augmente l'indépendance entre les interfaces. Si le pas de temps est trop grand, la fonction de filtre de la capacité peut être perdue, et on obtient des influences importantes entre les interfaces. Cela peut considérablement déranger l'algorithme de résolution.

Comme le montre cet exemple, il peut être justifié de négliger quelques termes dans le système global $\boldsymbol{C}$ à cause d'une influence mineure sur les interfaces. Pour cet objectif, il est nécessaire que le module contienne une capacité interne pour retarder le flux d'information qui le traverse. Dans le domaine de

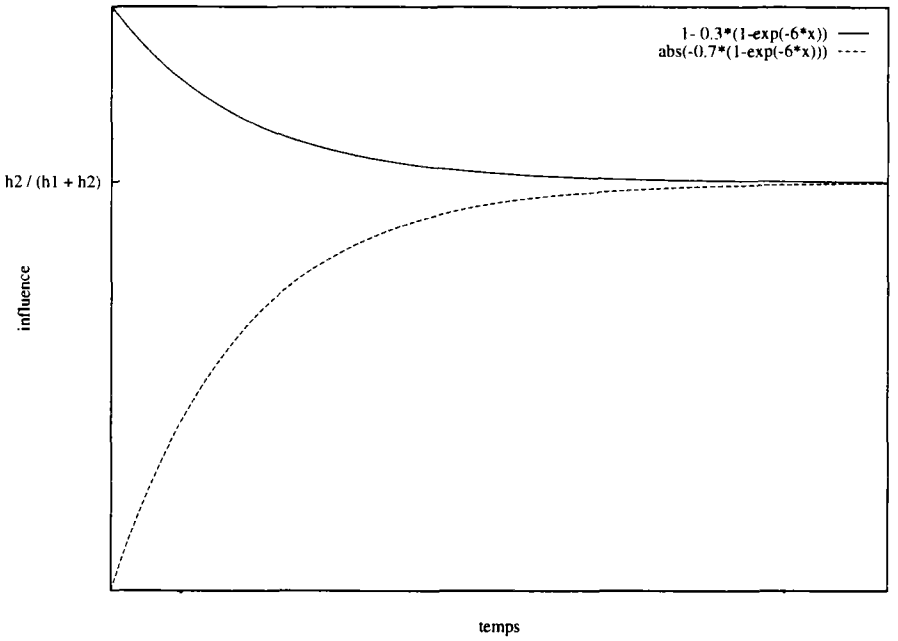


FIG. 6.6 - *L'influence des températures T_1 et T_2 sur le flux ϕ_1.*

la thermique, les modèles capacitifs sont très courants et dans beaucoup de cas, nous pouvons donc appliquer cette méthode LOCALE.

Nous obtenons quelques avantages, mais aussi des inconvénients au niveau calcul par l'approche de la résolution LOCALE des contraintes d'un module composé.

1° Les systèmes d'équations à résoudre deviennent beaucoup plus petits. Au lieu de traiter par exemple une matrice de taille cent fois cent (10.000 éléments), nous avons dix matrices de taille dix fois dix (1.000 éléments) à résoudre.

2° Dans beaucoup de cas, les calculs sont accélérés. Suite aux systèmes d'équations plus petits, il y a moins d'appels aux sous-modules dans l'algorithme de la résolution.

3° Les éléments négligés ont malgré tout une influence sur la solution. Une matrice *Jacobienne* correcte fait converger plus rapidement (en nombre d'itérations) que celle dans laquelle des éléments ont été supprimés. Cela entraîne des itérations supplémentaires, car les réponses des frontières changent.

4° L'ensemble des systèmes d'équations est numériquement moins stable que l'approche GLOBALE pour les mêmes raisons que le point précédent. En cas de problème de convergence, il faut plutôt choisir la méthode GLOBALE.

Outre les approches GLOBALE et LOCALE, nous avons également des algorithmes différents selon la technique de parallélisation des calculs de simulation. Les paragraphes suivants développe cet aspect.

6.4 Modules actifs : problème de la synchronisation

Comme expliqué dans le chapitre 4.4, nous supposons une architecture MIMD (au moins virtuelle) sur laquelle notre programme de simulation Motor-2 tourne. Si la machine physique n'a qu'un seul processeur, la simultanéité des exécutions des différentes parties du programme est réalisée par une couche inférieure du programme[1]. À chaque sous-module, nous avons associé sa propre file d'exécution. C'est à dire, chaque module terminal, mais aussi chaque module composé a virtuellement son propre processeur, ses propres données, en fait son propre programme, ou plutôt ce que l'on appelle sa propre tâche. Tous les modules peuvent s'exécuter indépendamment des autres.

La tâche des modules terminaux est spécifiée par leur modèle. La spécification du modèle contient toutes les actions de base, et notamment la capacité à calculer les réponses dans les frontières. Ceci est également valable pour le modèle des modules composés. Les modules composés contiennent essentiellement les interfaces avec les algorithmes de résolution de ses systèmes d'équations.

Comme les modules ont tous leur propre programme et avec cela leur propre avancement dans les calculs, il faut prévoir des *synchronisations* et des *échanges de messages* entre les objets de calcul séparés. Dans le langage de programmation Ada les deux objectifs sont réalisé par des *rendez-vous.*

Les rendez-vous ont toujours lieu entre deux tâches. Une tâche appelle une autre. Plusieurs points de rendez-vous peuvent être prévus pour la communication. Ils sont également appelés *points d'entrée.* Un appel au point d'entrée d'une tâche fait suspendre l'exécution de la tâche appelant. Elle attend un signal de la tâche appelée. Ce signal est envoyé quand la tâche appelée n'a rien d'autre à faire et s'il n'y a pas de rendez-vous à priorité supérieure à servir. Les deux files d'exécution échangent éventuellement des données et elles se séparent de nouveau. Les rendez-vous sont donc des instants de synchronisation entre les files d'exécutions indépendantes.

6.4.1 Points d'entrée aux modules actifs

En vue d'implanter un mécanisme de communication, nous avons donc spécifié des points d'entrée aux tâches par les rendez-vous. La vue extérieure d'un module est définie dans le chapitre 5.1.1. Nous y mentionnons les actions qu'un module peut exécuter. Ces actions sont accessibles comme des points d'entrée, c'est à dire des rendez-vous avec une tâche.

Les rendez-vous essentiels auxquelles toute tâche peut et doit répondre sont définis pour un modèle de base. Ce modèle est la racine obligatoire à partir de laquelle s'effectue la dérivation d'un nouveau modèle. De cette manière, la communication essentielle minimale entre les modules est assurée.

1. Le langage Ada contient le parallélisme dans sa définition. Tous les compilateurs livrent cette couche de parallélisme simulé.

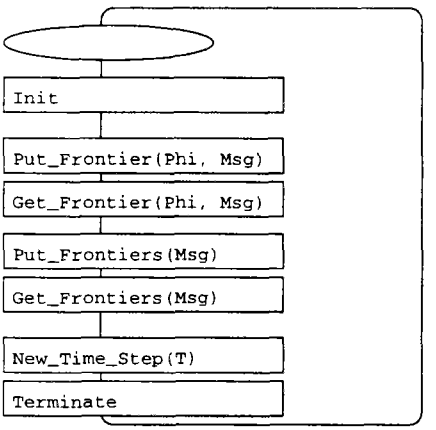


FIG. 6.7 - *Les entrées pour les actions de base des modules.*

La figure 6.7 montre une liste partielle des entrées de rendez-vous pour les modules. Les points d'entrée mentionnés ont les objectifs suivants. Nous indiquons par des flèches la direction d'échange de données pendant le rendez-vous. Une ↑ signifie un transfert d'information de la tâche appelée retour à la tâche appelant. Une ↓ envoie une information à la tâche appelée.

`Init(↑Msg)` Cette première entrée est nécessaire pour l'initialisation des interfaces avant le premier pas de la simulation. Les interfaces des modules composés demandent aux sous-modules ce qu'ils attendent comme valeur initiales aux pattes d'entrée. Ces valeurs peuvent être utilisées par les interfaces pour calculer l'état initial.

`Get_Frontiers(↑Msg)` Par ce point d'entrée, on peut récupérer les valeurs de toutes les pattes (de toutes les frontières) (`↑Msg`) d'un module par un seul appel. L'information obtenue n'est pas separée en frontières, mais elle a toujours cette structure.

`Put_Frontiers(↓Msg)` Pour envoyer des valeurs à toutes les pattes d'un module (`↑Msg`), un module composé utilise ce signal.

`Get_Frontier(↓Phi,↑Msg)` Pour une frontière spécifique qui est indiquée par `Phi`, le module composé peut interroger le sous-module. Il reçoit les valeurs des pattes (`↑Msg`) qui appartiennent à cette frontière.

`Put_Frontier(↓Phi,↓Msg)` C'est l'entrée inverse de la précédente. La tâche appelant envoie des valeurs des pattes (`↓Msg`) à une frontière spécifique (`↓Phi`) de la tâche appelée.

`New_Time_Step(↓T)` Si le module composé supérieur veut passer au nouveau pas de temps, il passe le nouvel instant dans (`↓T`). Ce signal est distribué récursivement par les modules composés aux modules terminaux.

`Terminate` Plutôt que d'arrêter le programme, nous envoyons ce signal. Il passe également par voie récursive jusqu'aux modules terminaux qui peuvent

encore entreprendre des actions comme la fermeture des fichiers avant de « mourir ».

Les points d'entrée ont deux objectifs. D'une part, ils servent comme canaux de communication puisqu'on peut échanger des données entre les tâches impliquées. D'autre part, ils sont des endroits de synchronisation entre la tâche appelant et la tâche appelée. Deux actions typiques de l'approche orientée objet ont dues être déclarées explicitement : l'initialisation et la terminaison de l'objet, parfois appelées aussi *constructeur* et *destructuer*.

6.4.2 Communication synchrone

Les points d'entrée sont utilisés pour établir la communication entre l'algorithme de résolution dans les interfaces et les sous-modules. Nous ne pouvons pas avoir une exécution continue dans toutes les tâches du système. Parfois les exécutions doivent être suspendues pour attendre des résultats qui sont calculés par d'autres modules. Les interfaces ne peuvent pas se mettre à calculer sans que tous les sous-modules aient terminé. Un exemple des phases actives d'un module composé et de ses sous-modules est visualisé dans la figure 6.8.

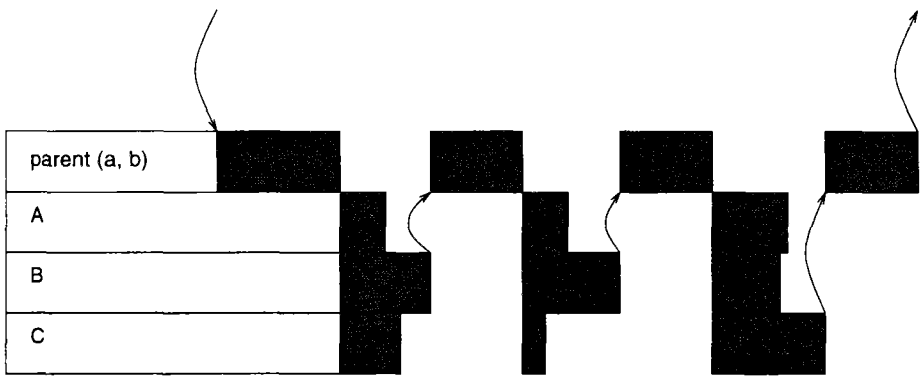


FIG. 6.8 - *L'enchaînement dit* synchrone *dans un module composé.*

Le module composé (marqué dans la figure comme *parent(a,b)*) reçoit un signal avec des nouvelles valeurs de pattes. Il doit re-calculer son état interne pour pouvoir répondre à un autre signal qui veut récupérer les valeurs de sortie. Comme nous avons vu, l'état interne d'un module composé est l'état des interfaces. Il initialise alors son algorithme interne et les interfaces *a* et *b*. Toutes les entrées des sous-modules sont connues. Elles sont donc envoyée par une commande `Put_Frontiers(↓Msg)`. Toutes les valeurs de toutes les frontières sont distribuées par module. Les sous-modules sont activés tous à la fois. Ensuite le module composé se met à attendre les réponses. Les sorties des sous-modules sont récupérées par la commande `Get_Frontiers(↑Msg)`. Comme pour les entrées, nous échangeons également toutes les valeurs de sortie par module à la fois.

Quand le dernier des sous-modules a terminé ses calculs, le module composé peut reprendre son activité. Il distribue les sorties des sous-modules aux inter-

faces concernées qui à leur tour peuvent appliquer leurs algorithmes internes. Une interface de type CONTACT PARFAIT, par exemple, calcule la somme des flux de chaleur reçus. Le résultat des calcul pour toutes les interfaces est soumis à l'algorithme global de résolution du module composé, si la méthode GLOBALE a été spécifiée (voir 6.3.1). Pour la méthode LOCALE (voir 6.3.2), toutes les interfaces appliquent l'algorithme de résolution indépendamment les unes des autres. Les nouvelles valeurs calculées dans les interfaces sont traitées par l'algorithme de couplage des interfaces. Pour une interface de type CONTACT PARFAIT, la nouvelle température est attribuée aux entrées des sous-modules. Un nouveau pas d'itération peut commencer.

C'est l'algorithme de résolution des interfaces qui décide quand les interfaces sont relaxées. Le module composé prend alors les sorties des sous-modules qui correspondent à ses propres frontières et il est maintenant prêt à répondre sur son propre point d'entrée qui demande les sorties (`Get_Frontiers(↑Msg)`).

Nous parlons ici d'une exécution synchrone, car les sous-modules ne sont appelés que quand toutes leurs entrées peuvent être envoyées. De plus, le module composé doit attendre le plus lent des sous-module avant de reprendre ses calculs. D'une façon synchrone seul le module composé *ou* les sous-modules sont actifs. Si les calculs sont distribués sur plusieurs processeurs, on peut donc garder à la fois le module composé et un des sous-modules sur un seul processeur pour mieux exploiter les ressources..

6.4.3 Traitement local avec enchaînement asynchrone

On peut pousser encore plus loin l'indépendance entre les sous-modules et les modules composés. Nous étendons l'idée de la méthode LOCALE et séparons entièrement les interfaces d'un même module composé. Non seulement elles font des calculs indépendants, mais elles ont également leur propre tâche, une exécution indépendante. Le module composé n'a qu'à lancer les tâches des interfaces et à distribuer et récupérer les valeurs pour ses propres frontières.

Nous supprimons la synchronisation des communications avec les sous-modules. Lorsqu'une interface veut entrer en communication avec un sous-module, elle le fait tout de suite sans attendre que d'autres données pour le sous-modules soient disponibles. Les fonctions que les sous-modules doivent livrer dans les frontières sont du type de l'équation 5.6 ($\boldsymbol{s} = \tilde{R}(\boldsymbol{e})$). Les sous-modules agissent seulement en fonction des excitations locales. Les réponses dans les frontières sont calculées par rapport à la variation des entrées locales en supposant que les autres entrées du module n'ont pas – ou seulement peu – changé. Nous exploitons ici le fait que les réponses dépendent plus des entrées locales que des autres entrées du module comme expliqué dans § 6.3.2.

Bien entendu, nous ne pouvons pas garantir la convergence de cette méthode. Encore plus que la méthode LOCALE en exécution synchrone, cette méthode asynchrone n'est pas stable. Elle fonctionne à condition que les perturbations dans les modules causées par l'algorithme de résolution ne soient pas trop importantes. Le calcul de la correction de l'itération NEWTON-RAPHSON

peut être basé sur des valeurs qui ne sont plus correctes quand la correction est appliquée. La convergence est certainement une fonction probabiliste, elle n'est pas garantie.

Pour la communication avec le sous-module, il faut passer l'identificateur de la frontière en question. Nous envoyons des valeurs d'entrée par le rendez-vous `Put_Frontier(↓Phi,↓Msg)` et nous récupérons les résultats avec la commande `Get_Frontier(↓Phi,↑Msg)`. Il n'est donc plus nécessaire d'envoyer toutes les entrées d'un sous-module, seules les valeurs d'une frontière sont échangées entre l'interface et le sous-module.

Comme les interfaces n'attendent pas que toutes les valeurs à envoyer soient disponibles, elles peuvent procéder et communiquer directement avec le sous-module concerné. Les sous-modules reçoivent donc des signaux de toutes les interfaces auxquelles ils sont raccordés. À chaque appel, ils se mettent à calculer. On voit ce comportement dans la figure 6.9.

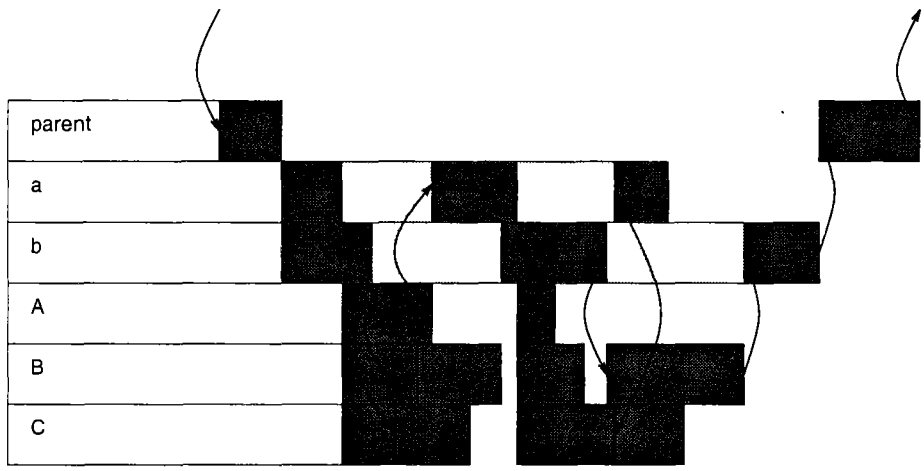


FIG. 6.9 - *L'enchaînement dit* asynchrone *dans un module composé.*

Le module composé indiqué par *parent* dans la figure, reçoit un signal et s'initialise (distribution de ses valeurs d'entrée aux sous-modules concernés). Ensuite les interfaces a et b entrent en activité. L'interface a termine son initialisation plus rapidement et envoie de nouvelles valeurs aux frontières des sous-modules A, B et C qui commencent les calculs. Le signal envoyé par l'interface b n'est traité par les sous-module B et C qu'après avoir répondu à a. Une tâche ne peut traiter qu'une question à la fois. On pourrait traiter ce problème par des « *acteurs* » qui se dupliquent (cf. [55]) et on pourrait aussi avoir un même module qui répond à plusieurs requêtes à la fois. On voit dans la figure que a reprend, quand A a terminé ses premiers calculs. L'interface b reprend quand B et C ont terminé. Dans le deuxième pas d'itération a déclenche de nouveau des calculs dans A, B et C.

Il faut remarquer dans ce mode de calcul qu'un sous-module accepte toujours les dernières entrées qu'il a vues dans ses frontières comme des entrées valides. C'est à dire que quand l'interface b déclenche les calculs dans B (dans la frontière

Φ_b^B), ce module prend les valeurs de la frontière vers l'interface a (Φ_a^B) de l'appel précédent comme entrées pour ses calculs actuels.

6.4.4 Connexions directes

Si l'ensemble des interfaces d'un module composé est constitué d'interfaces de type DIRECT, on peut appliquer un algorithme beaucoup plus efficace. Dans ce cas, il n'est pas nécessaire d'appliquer un algorithme de relaxation comme une méthode de NEWTON-RAPHSON. Toutes les valeurs de sortie peuvent être directement distribuées comme entrées aux modules suivants.

Le type d'interface DIRECT est courant dans la description des boucles ouvertes. Nous avons un enchaînement de modules ou l'un est appelé après l'autre. Prenons comme exemple la modélisation du rayonnement dans une enceinte où la température est imposée sur les surfaces. Pour déterminer la radiosité qu'une personne reçoit, nous plaçons des petits cubes à l'intérieur de la pièce. Cet exemple est traité dans les travaux de WURTZ [80]. Nous calculons maintenant l'éclairage des surfaces des cubes. L'algorithme suivant est appliqué :

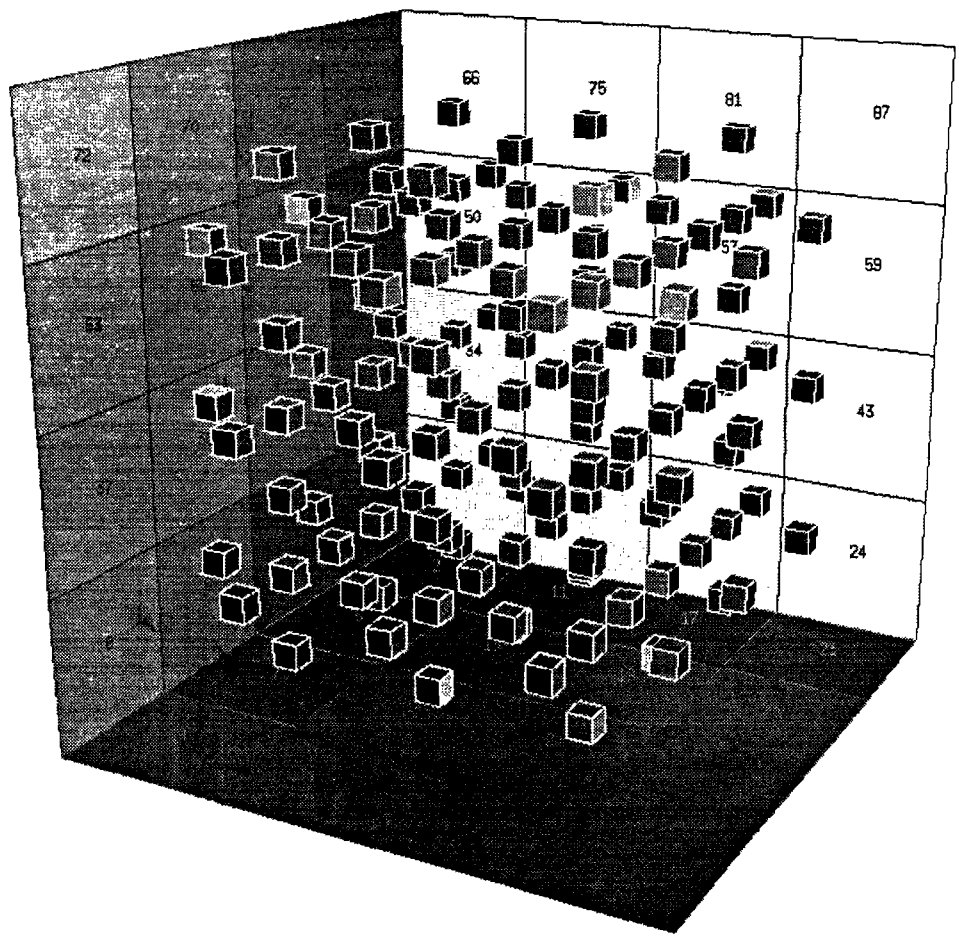


FIG. 6.10 - *l'éclairage des petits cubes sous rayonnement dans une enceinte*

- En fonction des températures sur les surfaces de l'enceinte, on calcule les radiosités des surfaces internes de l'enceinte
- Des interfaces DIRECTs passent ces valeurs aux cubes ...
- qui à leur tour calculent les éclairages des surfaces.
- Les résultats de tous les cubes sont passés par des interfaces DIRECTs à un module IMPRIMANTE ...
- qui les écrit dans un fichier.

La direction des dépendances impose un ordre naturel de calcul. La figure 6.10 montre le problème et dans la figure 6.11 on voit le découpage choisi.

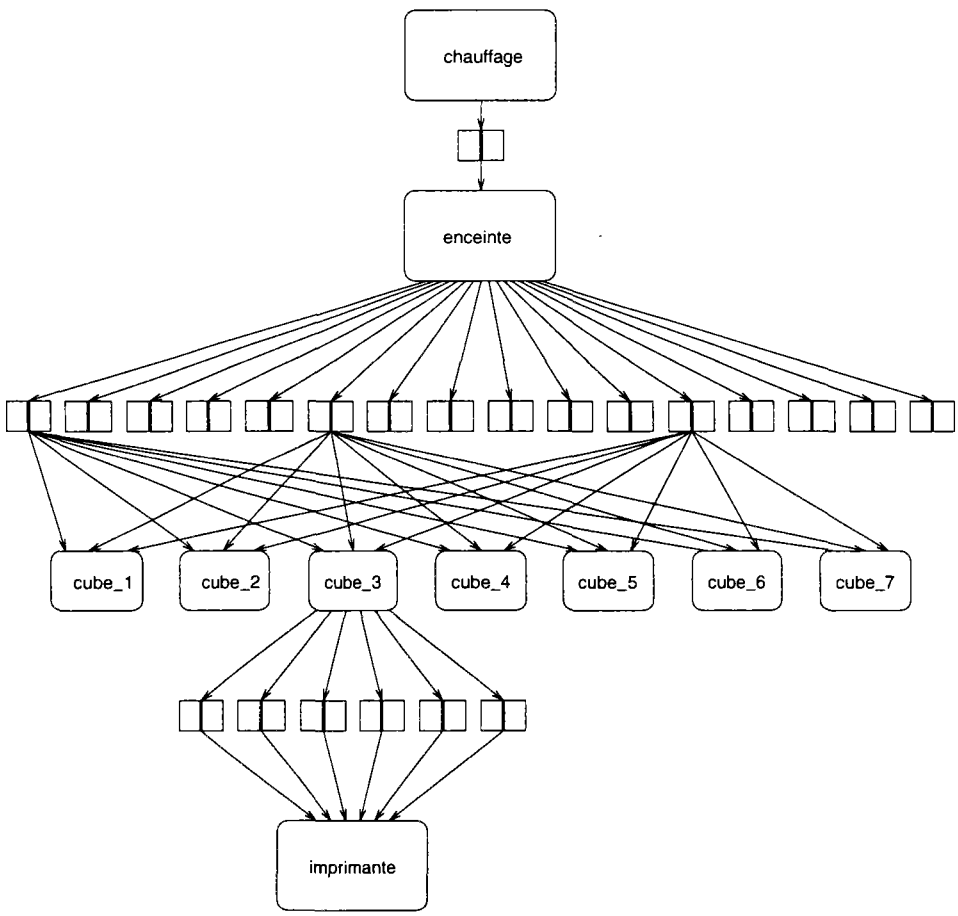


FIG. 6.11 - *Le découpage du problème. Pour ne pas trop charger la figure seules quelques connexions entre les sous-modules sont représentées. Les frontières des modules n'y figurent pas ; les rectangles avec tiret vertical représentent les interfaces du type* DIRECT.

Dans un module composé où tous les sous-modules sont connectés par des interfaces DIRECT on a intérêt à utiliser la classe des modules composés spécialisée pour ce processus. Dans une connexion directe la sortie du premier module

(A) sert comme entrée pour le deuxième module (B) sans action supplémentaire dans l'interface. Ce type de connexion est généralement (dans le module composé standard) décrit par une équation de simple égalité $\Phi_{if}^{A} = \Phi_{if}^{B}$. Mais cette équation n'exprime pas le caractère directionné du raccordement. (Dans un langage de programmation on écrivait plutôt $\Phi_{if}^{B} := \Phi_{if}^{A}$.) Avant de calculer le module B, il faut que le module A ait déjà calculé ses sorties, dont B dépend.

Dans ces cas spécifiques, il vaut mieux calculer un module après l'autre et cela dans l'ordre des dépendances. On commence alors avec le premier module. L'utilisateur doit spécifier dans le module composé lequel des sous-modules est considéré comme le premier. La suite des calculs est déjà implémenté dans l'algorithme actuel. Un algorithme ne devrait pas être trop compliqué qui détermine automatiquement le premier élément. Des algorithmes de traitement des graphes pour l'ordre de calculs sont par exemple implémentés dans Spark (voir § 2.3.4).

Pour toutes les sorties du module, l'interface correspondant lit les valeurs et distribue ces valeurs aux autres modules de l'interface. La radiosité de la première facette de l'enceinte est une entrée pour tous les sept cubes. On ne peut pas directement lancer les calculs du CUBE_1 car ce module nécessite aussi les radiosités des autres facettes. On traite donc d'abord toutes les interfaces qui sont concernées par les sorties du module actuel. Quand la dernière entrée d'un module a reçu sa valeur, on déclenche ce module et le traitement suivant de ses sorties. De cette façon, tous les sous-modules sont calculées récursivement jusqu'à ce que le module composé soit complété.

6.5 Modification de l'algorithme: interfaces avec « histoire »

Une tentative supplémentaire pour accélérer les calculs consiste à donner un état aux interfaces. Cet état stocke les valeurs des itérations précédentes. Il faut distinguer deux axes d'itération différents qui ont lieu pendant une simulation.

1° La discrétisation du temps conduit à faire une itération dans le temps pour avancer la simulation. Nous appellerons cette itération *horizontale.*

2° Nous devons résoudre des systèmes d'équations à chaque pas de temps pour chacune des interfaces. En général, c'est aussi un algorithme itératif, pour des raisons numériques. Nous l'appelons itération *verticale.*

La figure 6.12 illustre ces deux types d'itération.

La solution obtenue pour une interface évolue dans le temps en fonction de l'évolution de ses modules connectés. Habituellement, la valeur initiale de l'interface dans un nouveau pas de temps, est la solution obtenue au pas de temps précédent. Si l'on suppose que l'évolution de l'interface est continue et suit l'évolution des modules, on peut prendre une autre valeur initiale. Avant de commencer les itérations du nouveau pas de temps, nous extrapolons l'évolution

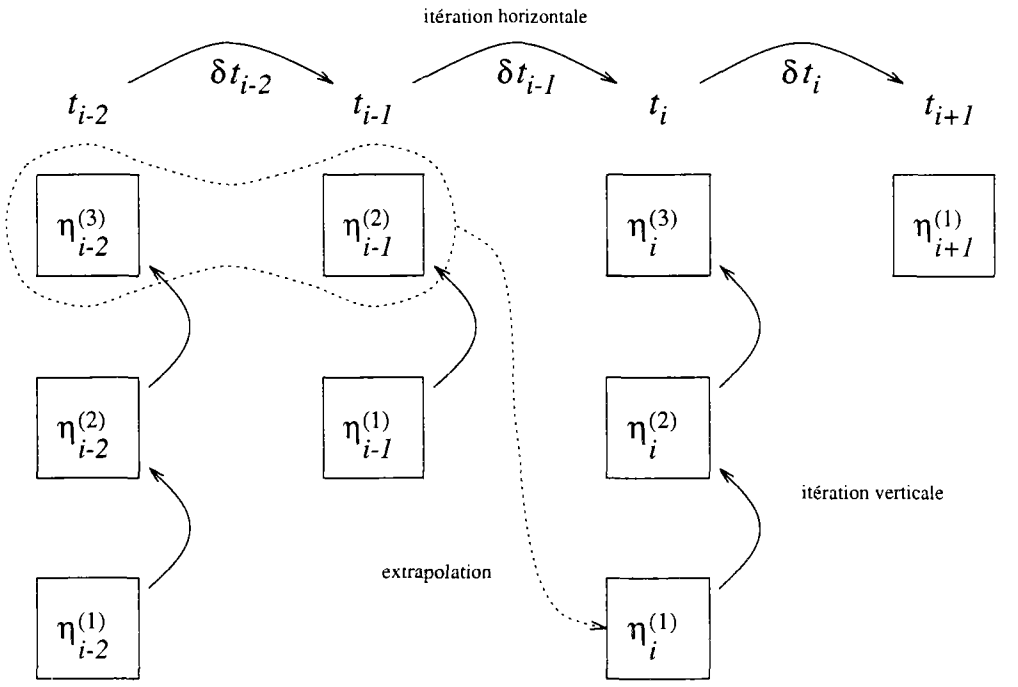


FIG. 6.12 - *Itération horizontale et verticale. L'avancement dans le temps se fait par une itération* horizontale. *L'état η_i à l'instant t_i devient δt_i plus tard η_{i+1}. Pour un instant donné t_i nous faisons une itération* verticale *pour résoudre les contraintes de connexion dans une interface.*

de l'interface vers le nouvel instant, et nous prenons cette valeur de l'extrapolation comme valeur initiale.

Cette nouvelle valeur de départ dans les interfaces est généralement beaucoup plus proche de la solution que la valeur au pas de temps précédent. Mais l'extrapolation nécessite que les interfaces stockent leurs valeurs à chaque pas de temps et aient donc une « *mémoire* ». Une condition supplémentaire est que les réponses des sous-modules n'aient pas de discontinuité au début du prochain pas de temps. Dans un tel cas la valeur de départ qui est extrapolée est aussi mauvaise que la valeur non-modifiée du pas de temps précédent.

6.6 Comparaison des méthodes

Il existent essentiellement quatre méthode possible de résolution dans l'environnement Motor-2. Toutes ces méthodes agissent au niveau d'exactement un module composé. En raison de structure hiérarchique de la description du système, les méthodes s'appliques alors recursivement au système entier.

La première méthode est la méthode GLOBALE. Elle resout l'ensemble de tous les systèmes d'équations constitués par les interfaces et les sous-modules comme *un* système d'équations complet. Toutes les dépendances entre les sous-modules sont prises en compte. Néanmoins, il existe une séparation de calculs.

Elle provient du découpage du systéme est la résolution indépendante par module. Même dans la méthode GLOBALE, il n'y a pas de communication directe avec les modules frères. Chaque module fait ses propres calculs, et c'est au prochain niveau supérieur dans un module composé, où sont vérifiées les contraintes de raccordement. Comme la méthode GLOBALE traite l'ensemble des équations, elle est numériquement la méthode la plus sûre.

Si l'on se repose sur un retardement du flux d'information à travers les modules, on peut également appliquer la méthode LOCALE. Chaque interface d'un module composé est traitée indépendament des autres. Si les modules communiquant avec deux interfaces contiennent des capacités, et si le pas de temps est assez court, les réponses locales ne changent pas tellement en fonction des calculs des autres interfaces. La résolution du systèmes d'équations de l'interface locale n'est donc pas dérangée en pratique par les événements des autres interfaces.

Cette séparation de calculs entre les interfaces peut être étendu encore plus, si l'on renonce aux appels simultanes des sous-modules. Chaque communiquation est traitée directement par un sous-module en question sans attendre les autres interface. C'est la méthodé ASYNC. La convergence ne peut plus être garantie. Mais pour les exemples traités dans ce mémoire, nous avons toujours réussi d'obtenir des résultats corrects avec cette méthode. Elle est la méthode la plus proche d'un système autoreglant, c'est à dire d'un système où chacun des composants « négocie » avec son environnement l'évolution de son état.

La quatrième méthode a un intérêt pratique, si les calculs des sous-moduls sont dans un ordre défini. La méthode DIRECT distribue les résultats d'un sous-modules aux entrées des sous-modules subséquants. Elle est surtout à applique dans les boules ouvertes ; pour les boucles fermées, l'utilisateur doit choisir un endroit où il ouvre la chaîne. Les deux bouts constituent l'entrée et la sortie d'un module composé (de type DIRECT) que l'on peut raccorder à une interface qui ensuite calcule la relaxation pour cette boucle.

Pour conclure, on peut dire que pour le cas général, la méthode LOCALE est à préférer. Le plus souvent elle est beaucoup plus rapide que la méthode GLOBALE (voir aussi les exemples au chapitre 8). Cette dernière est nécessaire en cas de problème numérique, par exemple pour les sous-modules avec discontinuités. C'est certainement la méthode la plus sûre. La méthode ASYNC est la plus souple de toutes les méthodes. Avec celle-ci, il est probablement le plus facile d'intégrer un pas de temps différent dans les différents sous-modules, car les calculs locaux ne sont pas contraints de suivre .... sans s'occuper du pas de temps interne des sous-modules. Il continuent de recevoir des réponses locales. La méthode ASYNC est à utiliser avec beaucoup de précaution au niveau numérique ; intellectuellement c'est la plus interessante.

Une particularité dans notre approche de simulation est l'intégration sur le temps qui se passe dans les sous-modules. Le programme Motor-2 ne l'effectue pas dans la résolution des interfaces. C'est donc le développeur de modèle qui doit appliquer un algorithme d'intégration dans sa description du modèle. Nous ne pouvons pas vérifier (dans Motor-2) que les algorithmes employés dans les

sous-modules sont compatibles entre eux[2]. Nous perdons de l'efficacité et de contrôle par cette façon de résolution. Mais l'alternative est d'implémenter un algorithme d'intégration dans les modules composés. Comme nous l'avons déjà dit, un tel algorithme nécessite des connaissance sur l'état interne d'un module. Ceci va à l'encontre de notre approche de visibilité limitée. L'état d'un module ne doit pas être visible à l'extérieur.

2. Il existe des travaux théoriques et pratiques sur la combinaison des « *solveurs* » communement appliqués aux même problème [5]

Chapitre 7

L'environnement Motor-2

7.1 Présentation

Dans le chapitre 3.2 déjà, nous donnions un aperçu du projet SYMBOL dans lequel le programme de simulation Motor-2 s'intègre. Comme d'autres environnements, Motor-2 n'est pas uniquement simplement un logiciel de simulation, mais il comprend aussi plusieurs utilitaires qui constituent *l'environnement de simulation Motor-2*. Le noyau de cet environnement est, bien entendu, le logiciel qui gère la simulation. Ce logiciel nécessite une description pertinente du système à simuler. Les composants de ce système sont des instances de modèles stockés dans une bibliothèque. Après la simulation, on veut généralement analyser les résultats générés. Ils sont donc à préparer par des filtres, des modifications, des calculs, *etcetera*. De façon schématique, l'environnement Motor-2 comprend donc les trois composants suivants :

1° Avant de pouvoir simuler un problème, il est nécessaire de le décrire dans un fichier particulier. Pour rester cohérent à travers tout le projet SYMBOL, un format de fichier qui contient la description d'un composant a été défini. Comme dans un formulaire, on spécifie le nom du module, ses pattes d'entrée et de sortie et les frontières typées qui relient les pattes. Outre ces informations, le créateur de ces fichiers doit donner un type aux composants ; il spécifie aussi le modèle de calcul qui est utilisé pendant la simulation. Selon le type de composant, il doit également donner des valeurs aux paramètres.

2° Le type qui est spécifié dans la description d'un module doit être reconnu par le programme de simulation. Nous avons établi une collection de modèles de calcul dans une bibliothèque. Mais une seule bibliothèque n'est pas suffisante. Non seulement les composants, mais aussi les types d'interfaces nécessitent un modèle de calcul. De plus, ces modèles sont exprimés aux différents niveaux d'abstraction. Au niveau informatique une bibliothèque est constituée d'une collection de fichiers contenant du code source. Les bibliothèques aux niveaux mathématiques et physiques n'existent pour l'instant qu'en partie.

3° Le post-traitement d'une simulation comprend l'analyse de résultats, mais aussi celle du déroulement de la simulation même. Plusieurs modules de visualisation sont disponibles pour l'affichage et l'impression des valeurs obtenues. Deux autres utilitaires permettent de suivre en détail l'activation et la terminaison des tâches.

Ces trois composants principaux de l'environnement Motor-2 (description, bibliothèques, post-traitement) sont représentés dans la figure 7.1. Le quatrième composant de la configuration est traité en annexe. Cette figure montre un agrandissement partiel du projet SYMBOL (voir la figure 3.1).

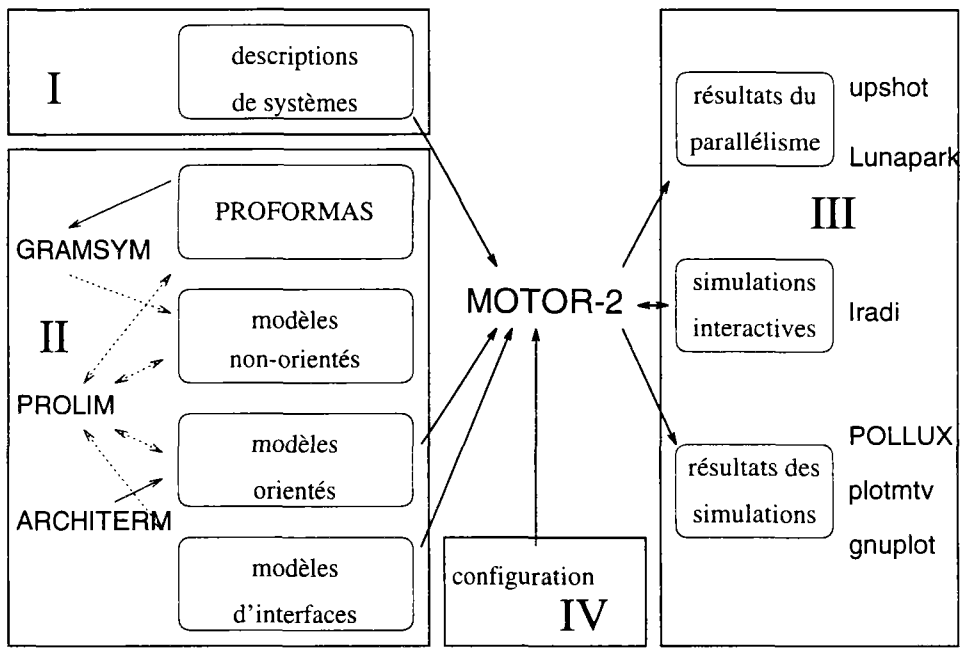


FIG. 7.1 - *Les composants de l'environnement Motor-2: l'ensemble I permet la description de système, l'ensemble II contient les modélothèques, l'ensemble III est dédié au post-traitement d'une simulation. La configuration et l'installation de Motor-2 sont réalisées dans l'ensemble IV.*

7.2 Description d'un système à simuler

En partant d'un découpage de problème, nous créons l'arbre de dépendance comme on l'a vu dans le chapitre 4.2. Chaque nœud (c'est à dire, chaque embranchement et chaque feuille) est représenté informatiquement par un fichier. Le format de ce fichier a été défini afin de permettre une description cohérente dans tous les programmes dans le projet SYMBOL. Il donne une vue extérieure sur le module dont le fichier est la représentation.

Le fichier contient le nom du module et un commentaire descriptif en langage naturel. C'est l'information minimum nécessaire pour la création d'un module et la gestion d'un ensemble de modules qui constituent le système. La déclaration

des pattes d'entrée et de sortie spécifie des noms, des types, les valeurs initiales et les unités physiques. Avec ces information, le programme Motor-2 peut vérifier la pertinence des raccordements. Les lignes suivantes définissent les frontières. Une frontière est composée d'un nom, d'un type et d'une liste de pattes (voir § 5.2). Ensuite on trouve le mot-clé qui spécifie le type du module. Les modules composés par exemple sont indiqués par le mot-clé `LinkUp`. Viennent ensuite les paramètres du modèle. Cela peut être une simple valeur comme pour le modèle COEF (le coefficient d'échange), ou la description des connexions internes d'un module composé. La syntaxe des fichiers en format SYMBOL est détaillée dans le mode d'emploi de Motor-2 (annexe B.1).

L'exemple de la figure 7.2 montre le texte d'un fichier qui représente le module W_WALL. Nous trouvons deux entrées (`Temp_Inside` et `Temp_Outside`) du type `Temperature` et deux sorties (`Flux_Inside Flux_Outside`) de type `Heat_Flux`. Dans les frontières (`Inside` et `Outside`) les pattes qui se correspondent topologiquement sont rassemblées. La partie inférieure qui suit le mot-clé `LinkUp` décrit une interface interne au module. La frontière `Inside` du sous-module W_INSUL est en contact parfait (`Perfect_Contact`) avec la frontière `Outside` du sous-module W_CONC. Les dernières lignes du fichier contiennent les équivalences entre les frontières du module composé et les frontières des sous-modules. La frontière `Inside` du sous-module W_CONC constitue la frontière `Inside` de W_WALL. De la même manière `Outside` de W_INSUL est la frontière `Outside` de W_WALL. La figure 7.3 montre la composition schématique de ce module.

Une étude de simulation contient une description complète d'un système à l'aide de ces fichiers. Le nœud supérieur est fourni à Motor-2 qui lit récursivement toute la structure. Typiquement tous les fichier de description se trouvent dans un seul répertoire. Commencé avec le fichier qui contient le module composé principal, ensuite les modules composés intermédiaires et finalement les modules terminaux, tous ensemble, ils constituent un projet. Pour effectuer une simulation, quelques autres informations sont encore nécessaires. Celles-ci concernent d'une part la configuration du simulateur (choix de méthode numérique, précision souhaitée, etc.), et d'autre part, le temps de la simulation (instants de début et fin, pas de temps fixe ou variable, valeur initiale). La spécification détaillée de ces informations se trouve également dans l'annexe B.

7.3 Bibliothèques

La description des différents objets d'un système (modules et interfaces) fait référence aux modèles dont l'objet est une instance. Il paraît naturel de stocker et collectionner ces modèles dans une *bibliothèque* quelque fois appelée *modélothèque*.

Comme une vraie bibliothèque de livres, une bibliothèque de modèles est un vecteur de capitalisation du savoir et de diffusion des connaissances. Elle permet l'accès aux différents composants créés en respectant la contrainte de

```
--                              -*- Symbol -*-
-------------------------------------------------------------------------------
-- This module 'W_Wall' represents the western wall of the
-- twodimensional room example. 'W_Wall' itself is built of the
-- submodules 'W_Conc' and 'W_Insul'.
-------------------------------------------------------------------------------
| 4.2                | Version
| Hand               | Origin
| W_Wall             | Model Name
>-----------------------------------------------------------------------------
--                              <<  Input  >>
--       Name         |        Type         |   Vdefault   |        Unit        |
-------------------------------------------------------------------------------
| Temp_Inside        | Temperature         | 19.00        |        C           |
| Temp_Outside       | Temperature         | 19.00        |        C           |
>-----------------------------------------------------------------------------
--                              <<  Output  >>
--       Name         |        Type         |   Vdefault   |        Unit        |
-------------------------------------------------------------------------------
| Flux_Inside        | Heat_Flux           | 0.000        |        W           |
| Flux_Outside       | Heat_Flux           | 0.000        |        W           |
>-----------------------------------------------------------------------------
--                              <<  Observ  >>
--       Name         |        Type         |   Vdefault   |        Unit        |
-------------------------------------------------------------------------------
>-------------------------------------------------------------
--                              <<  Frontiers  >>
--      Name          |        Type         |    Pins            |
-------------------------------------------------------------
| Inside             | First_Kind          | Temp_Inside        |
|                    |                     | Flux_Inside        |
| Outside            | First_Kind          | Temp_Outside       |
|                    |                     | Flux_Outside       |
>-------------------------------------------------------------
LinkUp
-------------------------------------------------------------
--     Type          |    Modul Name       |    Frontier        |
-------------------------------------------------------------
| Perfect_Contact    | W_Insul             | Inside             |
|                    | W_Conc              | Outside            |
>-------------------------------------------------------------
-- Father's Frontier |   Son's Name        |  Son's Frontier    |
-------------------------------------------------------------
| Inside             | W_Conc              | Inside             |
| Outside            | W_Insul             | Outside            |
>-------------------------------------------------------------
```

FIG. 7.2 - *La représentation informatique du module* W_WALL, *un fichier au format SYMBOL.*

modularité. Souvent on parle d'une unique modélothèque qui est capable d'accueillir tout type de modèle. Compte tenu de la distinction entre modèle de composant (pour les modules) et modèle de couplage (pour les interfaces), nous avons conçu plusieurs bibliothèques. Pour mieux profiter de la modularité et de la description traduite aux différents niveaux d'abstraction, il existe donc plusieurs bibliothèques correspondantes. Ces bibliothèques spécialisées traitent plus efficacement les modèles qu'elles contiennent.

Le plus souvent un modèle est paramétré. Différents objets peuvent alors être décrits par un seul modèle en fixant des paramètres à des valeurs différentes.

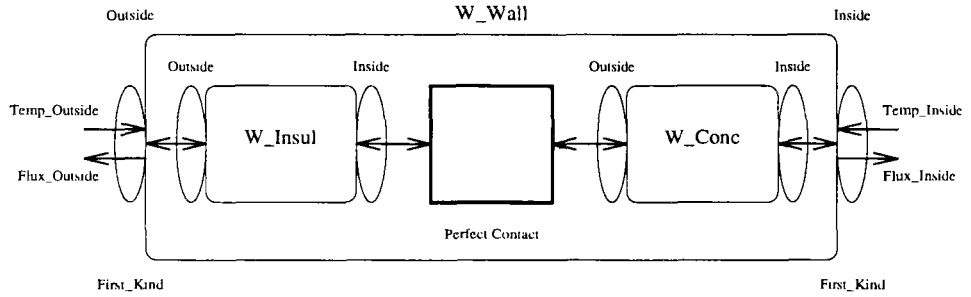


FIG. 7.3 - *Le schéma du module composé* W_WALL *comme il est décrit par le fichier de la figure 7.2.*

Le fait de laisser des paramètres libres permet de décrire plusieurs objets qui diffèrent par les caractéristiques associées aux paramètres. Cependant, ils restent semblables pour tout le reste, comme par exemple la structure géométrique, la loi interne de comportement, etc. On dit qu'un modèle est *générique* s'il comprend des paramètres libres. On crée une *instanciation* du modèle générique en donnant une valeur à un ou plusieurs de ces paramètres. Une instanciation complète consiste à donner une valeur à tous les paramètres d'un modèle, ce qui crée un module décrivant un composant du système. Si quelques paramètres restent libres, on parle d'une instanciation *partielle*. À chaque instanciation partielle, un nouveau modèle est créé, plus particulier que son modèle générique dont il est dérivé.

Cette démarche de spécification fait irréversiblement perdre une partie de la généralité qu'avait le modèle de départ. On a donc intérêt à abstraire au maximum les modèles développés en les construisant les plus génériques possible. C'est un objectif de modélisation qui ne peut souvent pas être atteint par souci d'efficacité ou par simple faisabilité numérique.

Une toute autre méthode de création de modèle est la dérivation par héritage. Dans le nouveau modèle, on fait référence à un modèle existant. On spécifie les extensions et les différences du nouveau modèle par rapport à l'ancien. Le modèle de *différences finies 1D* (FD1D1K) et le modèle d'un *coefficient d'échange* (COEF) peuvent être dérivés un modèle plus général qui est *échange monodimensionel 1k*. Dans le modèle *échange monodimensionel 1k* nous spécifions par exemple que tout modèle dérivé et toute instance de ces modèles a exactement deux frontières qui à leur tour sont du type *première espèce* avec une température d'entrée et un flux chaleur de sortie. La dérivation par héritage est un mécanisme très souple de la programmation orientée objet (voir § 4.3). Le concept se traduit naturellement pour la gestion de modèles.

Un problème qui généralement n'est pas résolu est l'*héritage multiple*. Un modèle peut être dérivé de deux ou plusieurs modèles différents en même temps. Il se pose alors le problème suivant : quel modèle ancêtre a la priorité, si des propriétés contradictoires sont héritées? C'est un sujet de recherche fortement discuté parmi la communauté du génie logiciel (pour la *programmation orientée objet*).

Héritage et instanciation sont deux modes complémentaires de dérivation de modèles. Les deux existent en même temps. La gestion d'une bibliothèque de modèle doit les prendre en compte et soutenir leur application. Traiter ce genre de modifications et garder en même temps les liens de dérivation montre les difficultés de conception de modélothèques. [1]

7.3.1 Représentation de modèles

Si un modèle depasse un certain niveau de complexité, on a besoin d'une description formelle avec une notation symbolique puissante pour exprimer aussi facilement le modèle que par l'écriture mathématique usuelle. Dans l'environnement Motor-2, nous avons appliqué une conception pragmatique qui nous a permis de réaliser rapidement un prototype fonctionnel. Il nous a fallu une représentation de modèle qui contienne au moins les informations nécessaires pour créer du code utilisable par un programme sur ordinateur. [2] Toute information supplémentaire soutient notre approche pour un environnement sûr et souple de simulation. Nous pouvions nous baser sur des travaux de DUBOIS [28], de CAPLAIN [7], et sur nos propres travaux [30]. Cela a abouti à une spécification des PROFORMAS. Un PROFORMA donne un cadre minimal pour la documentation des modèles de composant, du point de vue de l'utilisateur du modèle.

PROFORMAS selon DUBOIS

Dans le domaine de la simulation et de l'analyse énergétique, il y a beaucoup de recherches qui développent de nouvelles approches de modélisation et de nouvelles capacités de résolution. Pour faciliter le passage de ces nouvelles connaissances à leur forme appliquée à la simulation, les PROFORMAS ont été développés [28]. Ils sont une sorte de formulaire à remplir par le développeur d'un modèle. Les PROFORMAS fournissent une documentation précise, systématique et standardisée sur ses hypothèses et la validité du modèle. On y décrit non seulement les variables, les paramètres et les relations, mais aussi les règles sur l'usage et d'autres qualifications essentielles (plage de validité, par exemple).

De façon similaire à notre distinction en couches d'abstraction, les PROFORMAS décrivent aussi un modèle en plusieurs niveaux. Un PROFORMA est constitué de cinq chapitres qui contiennent :

1° une page initiale fournissant des informations essentielles pour l'identification du modèle (objet physique, problème thermique, méthode, ...)

2° une description formelle par des schéma, spécification des entrées, sorties, paramètres, variables et équations,

1. Les bibliothèques de composants sont encore un sujet de recherche ouvert dans beaucoup de laboratoires, y compris au sein du projet SYMBOL. Cette problématique concerne aussi bien la représentation et organisation des bibliothèques que la recherche et la réutilisation des éléments stockés. C'est également un problème dans le domaine de la modélisation (banque de donnée de modèles) que dans le domaine du génie logiciel (*software reuse*).

2. Le format de modèle NMF poursuit à peu près le même objectif. Il est envisagé d'intégrer également ce format dans le projet SYMBOL.

3° des clauses de validité et des règles d'usage pratique ; on trouve dans ce chapitre des règles sur la cohérence et sur la compatibilité qui sont utilisées dans la phase de connexion des modèles,

4° des rapports sur la validation du modèle (contexte, comparaison avec l'expérimentation, ...)

5° références scientifiques.

Une bonne cinquantaine de modèles sont déjà documentés au niveau européen. Mais remplir le formulaire est une tâche difficile. Une aide existe sous forme d'un logiciel qui guide l'utilisateur et qui propose un choix de mot et de classement par un thesaurus.

Les strucures de PROFORMAS présentées dans ce paragraphe sont encore volontairement informelles en quelques points (chapitre 3). Néanmoins elles sont déjà utilisées dans plusieurs projets du groupe ALMETH et au CSTB. Une version plus stricte du PROFORMA est utilisée par Gaz de France [49]. L'utilisation des PROFORMAS dans le projet SYMBOL pose des problèmes d'insertion informatique ; ceci a conduit à initier notre propre développement d'une grammaire de modèle (GRAMSYM) par CAPLAIN et LEFEBVRE.

PROFORMAS selon CAPLAIN – LEFEBVRE

Une grammaire a été développée pour permettre la description formelle de modèles sans perte de généralité. Sa réalisation est le logiciel GRAMSYM. Un modèle décrit dans le format GRAMSYM contient des équations, mais pas de méthodes de calcul. De même, un modèle peut encore faire intervenir des variables libres comme le temps t, qui doivent être ultérieurement discrétisées.

Le format GRAMSYM contient cinq parties. La documentation est un texte libre qui peut être proche de celui des PROFORMAS de DUBOIS. Outre les variables, chaque modèle comporte des paramètres, qui servent à obtenir des modèles plus spécifiques par instanciation. La notion de porte permet d'exprimer les relations de couplage entre les modèles. Le mode d'une porte peut déclarer la direction des communications comme entrée, sortie ou entrée-sortie non-spécifiée. Un modèle comporte par ailleurs des conditions de validité, expressions délimitant le domaine de valeurs des paramètres et des portes dans lequel le modèle physique décrit est valable. La dernière partie mentionne les équations sous forme implicite. Elles représentent un ensemble de relations entre des grandeurs, plutôt qu'une transformation d'entrées en sorties. Mais on peut aussi spécifier des versions explicites correspondant aux équations implicites.

Dans sa forme actuelle GRAMSYM réalise un certain nombre d'opérations. Il vérifie la cohérence de la description du modèle, notamment les équations et leur dimension physique. Ensuite il peut générer un fichier LaTeX contenant la description de modèle sous forme lisible. Une première analyse des équations peut être reprise par des logiciels de calcul formel tel que Macsyma ou Maple. Une description plus détaillée de GRAMSYM se trouve dans [51].

7.3.2 Modèles dans l'environnement Motor-2

La grammaire pour la description de modèle comme elle est définie par CAPLAIN et LEFEBVRE pas plus que les PROFORMAS de DUBOIS ne permettent pour l'instant de créer une représentation qui soit utilisable dans un programme de simulation. C'est pourquoi nous avons développé notre propre format, plus pragmatique, pour la description de modèles dans l'environnement Motor-2.

La représentation choisie est très proche de notre application pour la simulation. Le développeur de modèles exprime les paramètres et le comportement du modèle en termes informatiques. C'est à dire qu'il a à déclarer des variables et à écrire quelques lignes de code. Ceci ne doit pas être vu comme une limitation sur la généralité des modèles ou sur la liberté de les décrire. La bibliothèque de modèles Motor-2 se voit comme dernier élément dans une chaine de modèlothèques (l'ensemble II dans la figure 7.1). Nous supposons qu'il existe d'autres logiciels qui assurent le passage entre une représentation en format GRAMSYM vers notre format Motor-2.

Le format Motor-2 des modèles contient en principe quatre parties. Séparées par des mot-clef, on y trouve les sections suivantes :

1° un en-tête pour quelques informations administratives. Parmi ces informations il y a l'origine du modèle. Sa valeur est un modèle ancêtre d'où sont recupérées toutes les informations que l'on ne spécifie pas ici. Nous y reviendrons.

2° une partie optionelle peut déclarer les pattes et frontières standard pour les modules qui seront de ce type. Nous reprenons ici le même schéma de pattes et de frontières que pour le module. La définition des entrées et sorties au niveau de modèle plutôt qu'au niveau du module a un sens, si le nombre de pattes et de frontières est fixe pour toute instance du modèle. Un coefficient d'échange a toujours deux entrées et deux sorties ; un module de type PRINTER peut accepter un nombre d'entrées quelconque.

3° la déclaration des paramètres avec leur nom, leur type, et optionnellement des commandes pour la lecture et l'écriture. Ces paramètres libres sont à donner pour chaque module de ce type (cf. *Description d'un système* en haut).

4° le comportement dynamique, c'est à dire les équations et une description détaillée pour les résoudre sont à programmer dans cette partie. Pour l'instant il est nécessaire de savoir programmer les équations directement. Les programmes de passage des niveaux supérieurs ne sont pas encore opérationnels.

Dans ce format Motor-2, il n'existe pas de documentation sur le modèle, ni de limitations sur la validité. Aucun contrôle n'est effectué actuellement. Pour les raisons citées en haut, nous nous contentons du strict nécessaire pour l'environnement SYMBOL. Les vérifications et les informations supplémentaires sur les modèles, ainsi que la génération du code à partir de modèles en format

GRAMSYM sont situées dans les bibliothèques et passages plus élevés. Inscrire des commentaires sur la fonction du modèle est tout de même recommandé. Une description détaillée du format Motor-2 pour la création des modèles se trouve dans l'annexe B.2.

L'héritage

Comme déjà dit, un modèle Motor-2 peut faire référence à un autre modèle. Ce deuxième modèle est appelé *ancètre* du nouveau modèle qui à son tour est appellé un *modèle dérivé*. Il n'est pas nécessaire de spécifier tous les champs dans un modèle dérivé. Toutes les propriétés que Motor-2 ne trouve pas dans un modèle sont alors cherchées dans le modèle ancètre. Si elles n'y sont pas trouvées non plus, Motor-2 remonte récursivement jusqu'à l'origine de l'arbre de dérivation, le modèle GENERAL. Ce modèle de base ne spécifie aucun comportement. Si vraiment un modèle utilise des action qui viennent du modèle GENERAL, un message d'erreur est émis.

Puisque le problème de l'héritage multiple n'est pas résolu, on ne peut spécifier qu'un seul ancètre par modèle. La bibliothèque de modèles Motor-2 contient donc un graphe arborescante de dérivations de modèles. Une partie de cet arbre est visualisée dans la figure 7.4.

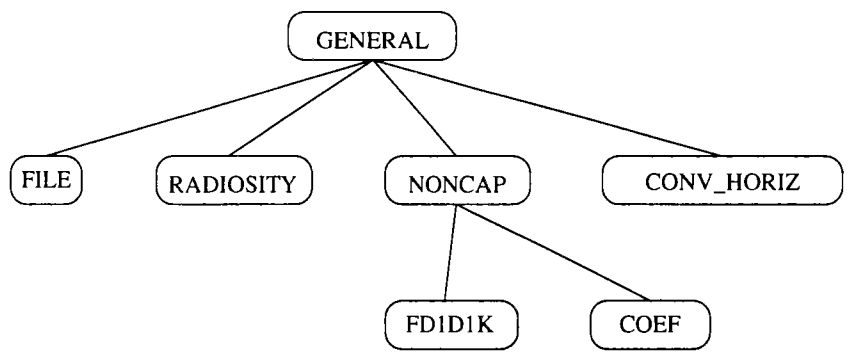


FIG. 7.4 - *Les dérivations de modèles dans la bibliothèque de modèle Motor-2.*

La conception des dépendances entre les modèles n'est pas une tâche évidente. Nous y retrouvons le problème de l'organisation d'une modélothèque. D'une part, ce problème concerne la maintenance. Le fichier descriptif ne contient pas la description complète, car on fait référence au modèle ancètre. Si le développeur veut vérifier quelques commandes, il doit chercher non seulement dans le fichier qui concerne directement le modèle, mais aussi dans toutes les descriptions de modèles dont il hérite des propriétés.

D'autre part, une bonne organisation des héritages diminue le montant d'informations qui est stocké dans la bibliothèque. On réduit la redondance. Cela permet, par exemple, que la correction d'une erreur dans un modèle de base, corrige cette erreur dans tous les modèles dérivés. Mais la modification est-elle vraiment souhaitée dans ces modèles dérivés? Souvent on devrait choisir *a priori* les relations entre les modèles, mais on ne sait pas encore quels seront les

modèles que l'on voudra inclure. Le classement pour un nouveau modèle doit rester assez souple et en même temps faciliter la création de nouveaux modèles.

Génération partielle de code

Quelques commandes ont été mises au point pour faciliter l'insertion d'un nouveau modèle dans la bibliothèque de Motor-2. Comme le logiciel de simulation ne traite pas que des équations, mais plutôt des algorithmes complets, ces algorithmes doivent être écrits d'une manière ou d'une autre dans un langage de programmation et ensuite être inclus dans le code exécutable de Motor-2.

Le développeur doit fournir un fichier en format « modèle Motor-2 » comme présenté en haut (§ 7.3.2) et détaillé dans l'annexe B.2. Les programmes de transition se chargent actuellement de générer toutes les déclarations nécessaires et tous les fichiers indispensables. Ceci concerne la déclarations des types et des fonctions de la lecture et de l'écriture. Elles sont utilisées pour les paramètres dans les fichiers de description de système (fichiers SYMBOL). Si le développeur n'a pas spécifié ses propres routines pour les entrées/sorties, elles sont générées automatiquement pour les types standard.

Pour le comportement dynamique des modèles, il faut déclarer des tâches et les inserer dans le contexte des modèles existants dans Motor-2. Nous utilisons des fichiers *squelette* qui sont remplis avec les informations provenant du fichier qui spécifie le modèle[3]. Avec un fichier de modèle correct, une seule commande est nécessaire pour le rendre disponible dans une simulation par Motor-2.

7.4 Post-traitement d'une simulation

L'exécution d'une simulation produit plusieurs sortes de sorties. Nous distinguons les résultats propres à la simulation, le mode interactif de Motor-2, et les résultats de l'observation du parallélisme.

1° Premièrement il y a les résultats de la simulation même. L'utilisateur a spécifié dans sa description du système des variables visibles dont il veut suivre l'évolution. Habituellement ces résultats sont stockés par des modules *imprimantes* (PRINTER) pendant la simulation.

2° Si les calculs de la simulation sont assez rapides, on peut faire communiquer Motor-2 avec un logiciel graphique. Dans l'environnement graphique, l'utilisateur peut modifier des variables d'entrées qui sont transmises à Motor-2. Les résultats de la simulations sont renvoyés immédiatement pour affichage dans l'environnement graphique. Ceci permet une étude interactive.

3. Par la voie de modification des fichiers squelettes, il existe la possibilité de reconfigurer Motor-2 pour qu'il n'utilise pas de parallélisme. Ceci est intéressant, si l'on veut un logiciel de simulation efficace sur une machine monoprocesseur.

3° L'exécution en parallèle des différents modules est parfois difficile à percevoir. Motor-2 est capable de représenter les différentes activations et suspension de tâches sous une forme plus lisible.

7.4.1 Résultats de la simulation

Plusieurs modèles de Motor-2 permettent de suivre l'évolution des variables. Le plus courant de ces modèles est PRINTER qui accepte un nombre quelconque d'entrées et qui les écrit dans un fichier. Un module de type PRINTER fonctionne comme tout autre module. Il faut le connecter par une interface à la frontière dans laquelle se trouve la variable à observer.

Le format du fichier créé par un modèle PRINTER est configurable par des paramètres lors de la spécification d'un module de ce type. On peut par exemple échanger l'ordre des données, normer les valeurs ou ajouter une colonne de données qui contient la valeur de l'horloge de la simulation.

Plusieurs utilitaires permettent de visualiser ces données ainsi créées. Gnuplot et Plotmtv sont deux logiciels du domaine public qui assurent très correctement cette mission. Ils peuvent lire les fichiers de données. Sur demande ils font quelques calculs internes comme un lissage ou des calculs de contours par exemple. Ensuite ils affichent à l'ecran de l'ordinateur les résultats sous forme de courbes d'évolution ou de champs d'isovaleurs. L'impression de ces courbes est également possible. Les images de ce mémoire ont été créées avec ces logiciels.

Ce type d'observation n'est possible que pour les variables qui ont été prévues dans cet objectif, c'est à dire qui sont rendues visibles dans une frontière. Les variables internes des modules ne peuvent pas être observées par d'autres modules. Ceci est un des objectifs majeurs de la conception orientée objet. Néanmoins, l'environnement Motor-2 inclut la possibilité d'envoyer des signaux aux modules pour qu'ils écrivent eux-mêmes leur état actuel dans un *fichier journal* (*logfile*). L'utilisateur peut spécifier si le signal est déclenché à chaque pas de temps ou à chaque appel du module. Avec des filtres, l'utilisateur peut ainsi récupérer toute variable du système.

D'autres types de modules permettant l'observation de variables sont traités dans le paragraphe suivant.

7.4.2 Motor-2 interactif

Parfois, il est utile de voir directement la réponse à une modification du système. Sous condition qu'un logiciel soit également capable de communiquer par des services du système d'exploitation, Motor-2 permet des simulation interactives.

Deux modèles spéciaux existent dans la bibliothèque Motor-2. Ils permettent de lire ou d'écrire des données dans des *pipes* ou des *sockets* d'UNIX[4]. Quand

4. Une *pipe* est un canal de communication avec d'autres processus qui tourne en même temps sur la même machine. Un *socket* établi la connection avec une autre machine dans un réseau.

Motor-2 reçoit de nouvelles données dans le canal d'entrée, il se met à simuler. Les calculs terminés, il renvoie les résultats dans le deuxième canal. Cette porte de communication peut être utilisée par un logiciel quelconque. Elle permet l'insertion facile de Motor-2 dans un environnement existant.

Ces deux modèles spéciaux sont utilisés dans le logiciel d'application Iradi. C'est un programme interactif couplé avec Motor-2. Il est dédié à un accès facile à l'étude du confort radiatif dans une pièce (voir [80]). La modélisation de ce phénomène a été expliquée dans le § 6.4.4. Dans le logiciel Iradi, l'utilisateur peut interactivement modifier les températures des surfaces de l'enceinte. Ces températures sont passées à Motor-2 qui calcule les éclairages et renvoie les résultats à Iradi. L'utilisateur peut donc voir en temps réel les effets des changements de températures sur l'échange radiatif dans la pièce. L'exemple d'un tel affichage est montré dans la figure 6.10 (page 95).

7.4.3 Observation du parallélisme

Généralement, on peut prévoir le comportement d'un programme séquentiel en lisant l'algorithme utilisé. Mais le comportement des programmes parallèles est très difficile à prévoir. De plus, des programmes parallèles présentent plus facilement des *erreurs de performance*, c'est à dire qu'ils livrent une réponse correcte, mais beaucoup plus lentement qu'attendu.

L'application du parallélisme dans un environnement de simulation pose le problème de l'observation. On veut savoir où la simulation met du temps, quels sont les modules actifs ou suspendus. Dans ce but deux utilitaires, Lunapark et Upshot, ont été intégrés dans l'environnement Motor-2.

Lunapark

Nous avons développé un utilitaire Lunapark pour mieux suivre une simulation parallèle réalisée à l'aide de Motor-2. Lancé sur la même machine que la simulation, Lunapark ouvre une fenêtre à l'ecran, dans laquelle il affiche symboliquement la structure hiérarchisée du problème à simuler. Chaque module composé et terminal est représenté par un petit carré. Les sous-modules d'un même module composé sont connectés par une ligne blanche. La figure 7.5 montre un exemple. Dès que la simulation commence, Lunapark fait clignoter les carrés qui représentent les modules actifs, les carrés des modules suspendus sont éteints. Grace à cette information, l'utilisateur peut déterminer les modules lents, ou les modules qui peuvent être coupés pour accélérer la simulation de l'ensemble.

Dans le logiciel Lunapark, on peut choisir un mode dans lequel la vitesse de la simulation n'est plus déterminée par les calculs, mais imposée par une horloge interne de Lunapark. C'est une façon de ralentir Motor-2 afin de mieux suivre l'ordre des activations. Chaque fois que Motor-2 signale à Lunapark un changement d'état des activités, l'exécution de Motor-2 est globalement suspendue.

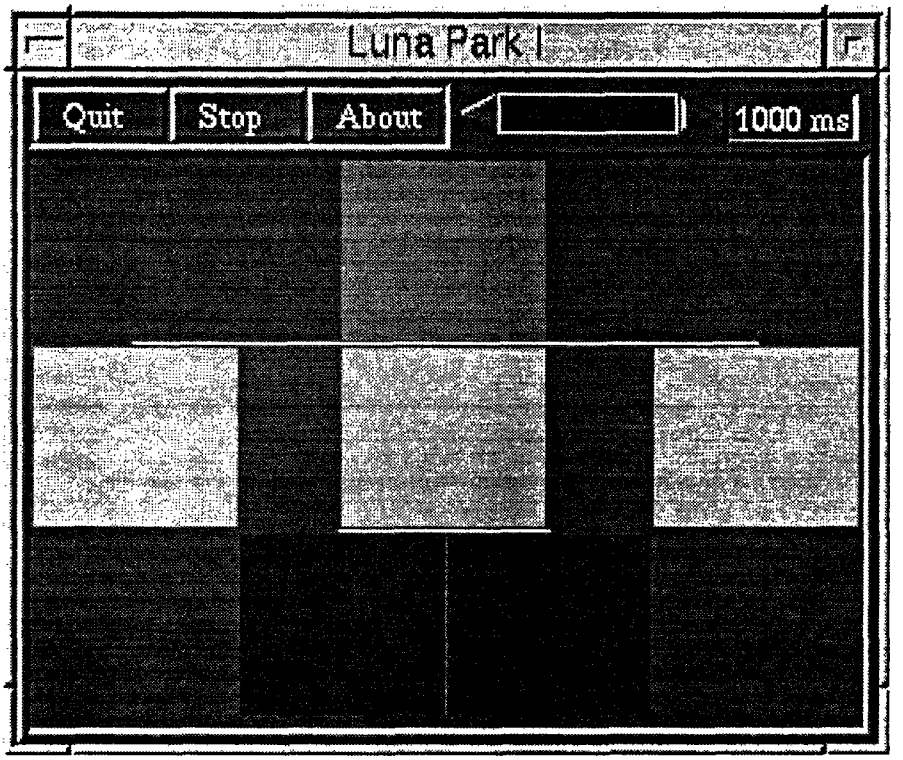


FIG. 7.5 - *L'affichage par Lunapark.*

Upshot

Généralement, les mises en activité et en suspension des tâches se produisent à une vitesse qui ne peut pas être directement observée à l'œil « nu ». Les ralentissements effectués par Lunapark font perdre les informations sur les durées relatives des activités des modules. Un deuxième utilitaire, Upshot – développé pour les grandes machines parallèles [45] – a été donc adapté à l'environnement Motor-2.

Motor-2 enregistre pendant la simulation toutes les activités relatives au parallélisme dans un fichier spécial. Ce fichier est analysé et présenté par l'utilitaire Upshot. C'est donc après la simulation, en « différé» qu'intervient l'analyse du comportement parallèle.

Si une simulation Motor-2 est configurée pour l'analyse par Upshot, les tâches écrivent au début et à la fin de chaque appel d'une communication des informations dans un fichier journal. Ce fichier contient donc une liste d'événements dans un ordre chronologique avec les identificateurs des modules, et les points d'entrée appelés, ainsi que les instants à la milliseconde près de ces appels. Upshot lit les informations et les affiche à l'écran, dans l'ordre temporel. La figure 7.6 en montre un exemple.

Upshot permet différents modes d'affichage pour les données enrégistrées. L'utilisateur peut choisir entre l'affichage des événements qui se sont produits et l'affichage par état de tâches. Pour ce deuxième mode, une définition des

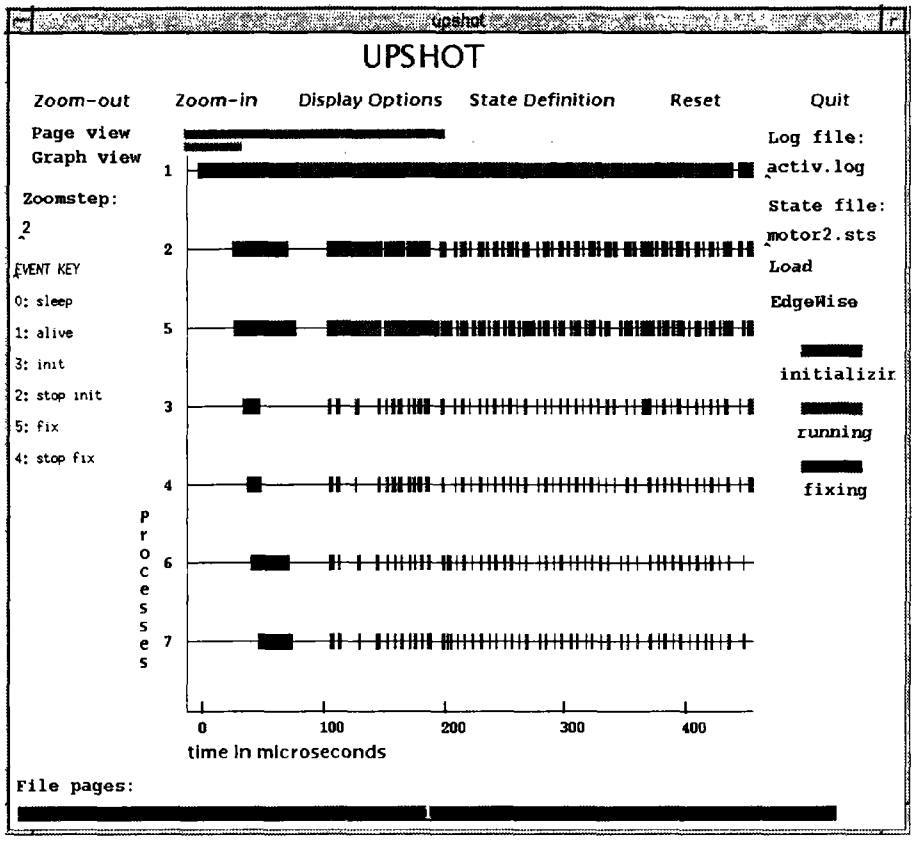


FIG. 7.6 - *L'affichage par Upshot.*

états a été créée. Les débuts et les fins des états sont définis par événements. Dans le cas de simulation avec Motor-2, l'affichage par état est le mode préféré. De cette façon, les phases actives des tâches deviennent directement visibles. Nous marquons en bleu la phase d'initialisation, en rouge des activités de calcul, et en marron les autres activités possibles (`Show` et `New_Time_Step`).

Chaque tâche du système est représentée par une ligne sur laquelle sont marqués les changements d'état par un changement de couleur. Upshot permet de « zoomer » l'axe de temps, c'est à dire de changer l'échelle temporelle en la réduisant ou l'allongeant. Une échelle plus fine permet la séparation des événements, une échelle plus grossière permet de reconnaître des régularités (ou irrégularités) dans la suite de l'activation des tâches.

Comme nous l'avons vu, l'environnement Motor-2 se veut plus vaste qu'uniquement un programme de simulation. Indispensable est son langage de description de problème. L'organisation par fichier renforce la modularité et des calculs hiérarchiques. Mais Motor-2 contient également des dispositifs pour la création et la gestion de modèles. Un post-traitement des résultats est rendu possible d'une part par des programmes spécialement développés comme Lunapark et Iradi et d'autre part par des programmes adaptés à cet usage (Plotmtv, Gnuplot, Upshot).

Chapitre 8

Exemples

Pour montrer l'usage de l'environnement Motor-2, l'application du découpage hiérarchique et la description de système aux différents niveaux d'abstraction, nous présentons trois exemples. Le *mur bicouche* est un exemple très simple, mais il permet de présenter l'application des concepts de base sans compliquer la démonstration par une physique complexe. Le deuxième exemple, celui de la *cellule de Hambourg* nous permet de valider le programme de simulation, puisque cet exemple a fait l'objet de comparaisons dans plusieurs environnements de simulation. Un dernier exemple – un local avec convection libre, échange radiatif et transfert conductif – montre l'application de Motor-2 à un problème non-trivial.

8.1 Mur bicouche

Tout au long de ce mémoire, nous utilisons l'exemple du mur bicouche pour présenter les différents aspects de l'environnement Motor-2. Ici, cet exemple est traité complètement et en détail.

8.1.1 Description du problème

Nous décrivons notre système à l'aide de la grille constituée des différents niveaux d'abstraction, tels qu'ils ont été présentés au chapitre 4.1. Il peut être utile de se reporter à la figure 4.1 (page 48) pour mieux suivre les étapes qui va d'une description technique au niveau algorithmique et à l'exécution de la simulation même.

Nous commençons au niveau technique. Le problème se décrit comme un mur en béton sur lequel est clouée une couche isolante en laine de roche, protégée par une couche d'aggloméré. C'est le cas typique d'un mur extérieur de bâtiment.

Au niveau physique, nous associons la conduction thermique à travers ce mur. La distribution des températures sur la surface ne nous intéresse pas ;

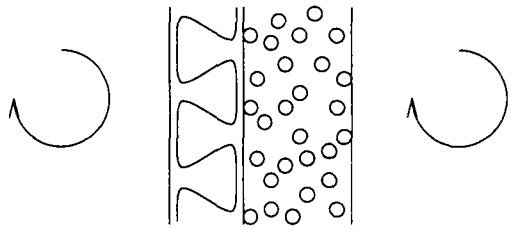


FIG. 8.1 - *Le schéma physique du mur bicouche.*

il est suffisant de considérer que ce problème a une seule dimension (dans la direction de l'épaisseur du mur). Le mur même a des échanges convectifs avec l'air intérieur et extérieur, et vérifie une loi avec coefficients d'échange. Pour simplifier cet exemple, l'aggloméré est négligé. La figure 8.1 montre la vue du niveau physique du mur. La géométrie de l'exemple est fixée à une surface de $10m^2$, l'épaisseur du béton est de $20cm$ et celle de l'isolation est de $8cm$.

On peut donc facilement distinguer quatre modules pour lesquels seulement deux *modèles* sont nécessaires. Les deux couches du mur (isolation et béton) avec capacité thermique sont des instances d'un modèle de conduction monodimensionnelle et les deux coefficients d'échange utilisent un modèle de résistance thermique. Entre les quatre composants nous supposons des interfaces qui ont tous le même type de couplage, un contact parfait. Les frontières des composants sont les surfaces où nous couplons les températures et les flux. On retrouve le schéma dans la figure 8.2

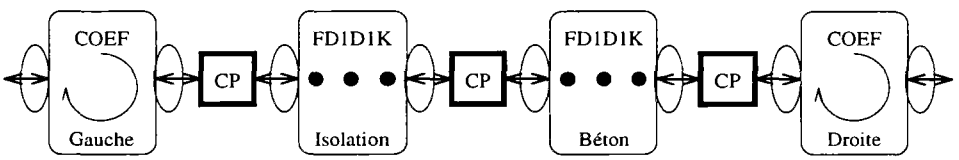


FIG. 8.2 - *Le schéma de découpage. Les quatre modules avec leurs modèles associés et les trois interfaces pour le raccordement.*

L'expression mathématique du modèle de conduction est la loi de FOURIER à une seule dimension :

$$\frac{\mathrm{d}T}{\mathrm{d}t} = \frac{\lambda}{\rho c}\frac{\mathrm{d}^2T}{\mathrm{d}x^2} \tag{8.1}$$

où λ est la conductivité, ρ la masse volumique, et c la capacité calorifique massique. La résistance thermique est une simple relation :

$$\phi = Ah\Delta T \tag{8.2}$$

avec h comme coefficient d'échange (l'inverse de la résistance thermique), S la surface d'échange, ΔT la différence de températures et $\phi = \frac{dQ}{dt}$ le flux de chaleur. Dans notre exemple concret, nous utilisons les valeurs caractéristiques thermo-physiques du tableau 8.3.

Au niveau algorithmique, nous devons résoudre les équations de l'étape mathématique. D'une part, il faut orienter les équations et d'autre part il faut les

conduction			*convection*		
	ISOLATION	BÉTON		GAUCHE	DROITE
λ $[\frac{W}{mK}]$	0.04	1.75			
ρ $[\frac{kg}{m^3}]$	30	2500	h $[\frac{W}{m^2K}]$	11	9
c $[\frac{J}{kg}]$	850	820			
A $[m^2]$	10	10	A $[m^2]$	10	10
l $[m]$	0.08	0.20			

FIG. 8.3 - *Les valeurs thermo-physiques des matériaux du mur, des coefficients d'échange et des caractéristiques de la géométrie.*

discrétiser. Nous choisissons de discrétiser en espace le modèle de conduction. L'équation de la chaleur s'exprime maintenant sous forme matricielle :

$$\frac{\partial \Theta}{\partial t} = \boldsymbol{A}\Theta + \boldsymbol{ET} \tag{8.3}$$

où nous désignons par $\Theta = [\Theta_1(t), \cdots, \Theta_N(t)]^t$ le vecteur d'état des températures dans les nœuds et $\boldsymbol{T} = [T_1, T_2]^t$ le vecteur qui contient les températures des bords. Les flux entrant et sortant aux bords sont exprimés maintenant par

$$\phi = \boldsymbol{J}\Theta + \boldsymbol{GT} \tag{8.4}$$

Nous créons dix nœuds dans l'épaisseur du composant. Après discrétisation du temps, nous pouvons appliquer un algorithme d'intégration de l'équation différentielle. Ensuite nous pouvons exprimer les flux aux bords en fonction des températures imposées. La méthode simple d'EULER donne

$$\Theta_{i+1} = \Theta_i + \Delta t(\boldsymbol{A}\Theta_i + \boldsymbol{ET}_i) \tag{8.5}$$

et le flux est donc

$$\phi_{i+1} = \boldsymbol{J}\Theta_{i+1} + \boldsymbol{GT}_i{}^1 \tag{8.6}$$

Nous avons donc les formules nécessaires pour recalculer l'état interne (le champ de températures) et les réponses des frontières (les flux) en fonction des valeurs entrantes des frontières (les températures aux bords). Dans la modélothèque de Motor-2 ces fonctions sont implémentées dans un modèle nommé FD1D1K. Ce nom est la version courte du nom complet FINITE DIFFERENCES, ONE DIMENSION, FIRST KIND. Ce nom évoque les modèles ancêtres de la bibliothèque qui sont

1° **ONE DIMENSION**. Ceci veut dire que toute instance de ce modèle a exactement deux frontières. Par défaut elles sont appelées LEFT et RIGHT.

2° **FIRST KIND**. Le type des frontières et l'orientation de leurs variables sont fixés aux frontières avec une température comme entrée et un flux comme sortie.

Les autres paramètres libres de ce modèle sont le nombre de nœuds internes et les valeurs thermo-physiques comme spécifiées dans la figure 8.3.

Les calculs des résistances thermiques ne nécessitent pas d'opérations internes. Il n'y a pas d'état interne. L'équation 8.2 s'applique directement. Le modèle dans la bibliothèque Motor-2 est COEF. Il est également dérivé de ONE DIMENSION et de FIRST KIND avec les mêmes conséquences que pour le modèle FD1D1K ; deux frontières avec températures comme entrées et flux comme sorties.

Les interfaces sont du type CONTACT PARFAIT. Elles contiennent et conservent une température qui est envoyée aux modules connectés. Les modules à leur tour répondent avec le flux. L'interface calcule la somme des flux et fait varier sa température pour annuler la somme de flux. Nous appliquons l'algorithme optimisé du chapitre 6.2.2 pour les interfaces du CONTACT PARFAIT.

8.1.2 Simulation et résultats

La première ligne de la figure 8.4 montre un module principal MUR_OUEST qui est un module composé de quatre sous-modules. Ceci est le choix le plus simple pour une hiérarchie de calcul ; tous les sous-modules se trouvent au même niveau. Trois interfaces internes imposent les conditions de raccordement. Nous appelons cette configuration le découpage **L**.

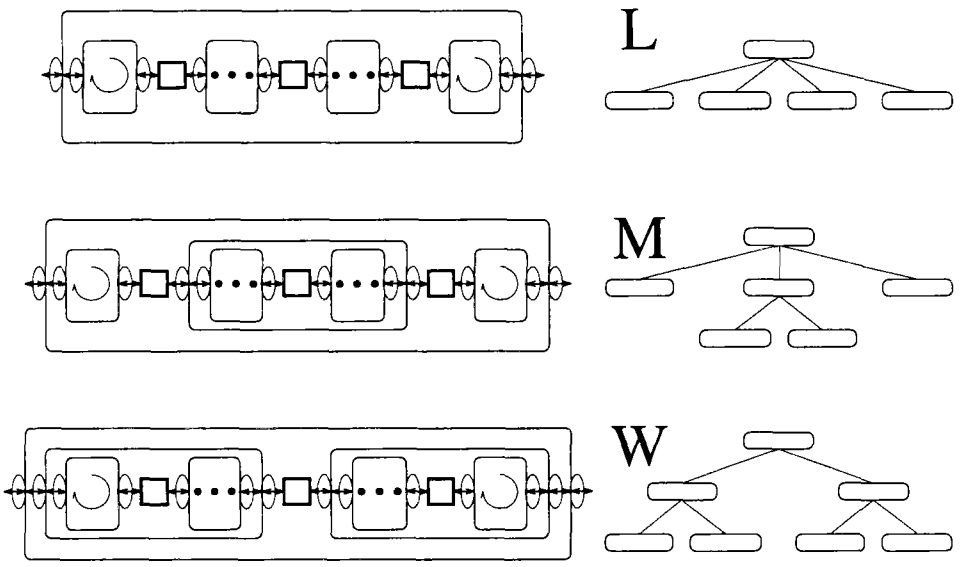


FIG. 8.4 - *Différents graphes de découpage et couplage possible avec les noms de référence.*

Mais nous avons testé également d'autres configurations comme schématisées dans la figure 8.4. La configuration **M** est un découpage plus « *physique* ». Les deux couches du mur constituent ensemble un module composé MUR avec une interface entre les modules terminaux. Les trois module EXT, MUR, et INT sont connectés par deux interfaces pour construire le module principale

OUEST_MUR_M. La configuration **W** est issue de considérations numériques. Comme l'évolution la plus rapide se trouve dans l'ISOLATION et le coefficient de EXT, ces deux modules sont mis ensemble pour éviter des activations non-nécessaires des modules de droite.

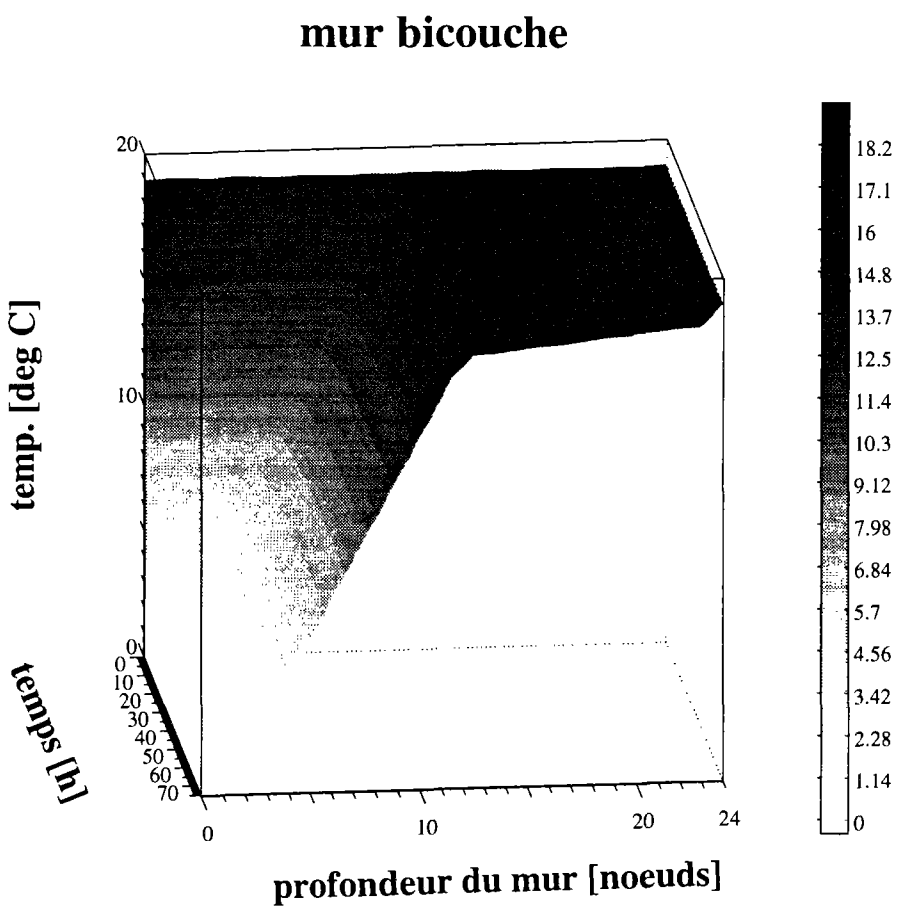


FIG. 8.5 - *L'évolution temporelle jusqu'au régime permanent du champ de températures du mur bicouche.*

La simulation effectuée à l'aide des configurations **L**, **M** et **W** est initiée avec une température uniforme dans tous les modules de $19°C$. À l'instant $t = 0$, la température extérieure (à gauche) tombe à $0°C$. La simulation est effectuée avec un pas de temps de 1800 secondes. La *Jacobienne* interne n'est réévaluée qu'après trois pas d'itération. Le seuil d'arrêt des itération est fixé à une précision de 0.01 %. La simulation est poursuivie jusqu'à l'état stationnaire. Il est atteint après 75 heures. Non seulement les différents découpages, mais aussi les différentes méthodes de résolution, ont été testés à l'aide de ces exemples.

L'évolution temporelle des température internes du mur est montrée dans la figure 8.5. La hauteur[2] indique la température. Au fond de la figure se trouve le

2. et les niveaux de gris qui sont mal visibles dans cette reproduction.

pas zéro où toutes les températures sont à $19°C$. Au premier plan, la distribution de température a atteint le régime permanent. La profondeur du mur est discrétisée en 25 nœuds. Le nœud zéro est la température imposée de l'extérieur, le nœud 24 la température imposée de l'intérieur. Les nœuds 1, 12 et 23 contiennent les températures des interfaces. La couche de l'isolant est représentée par les dix nœuds 2 à 11, et la couche de béton par les nœuds 13 à 22.

On remarque que la température qui tombe à zéro à l'extérieur au début de la simulation ne devient sensible pour la couche de béton qu'après environ 10 heures. Il faut attendre encore plus que 60 heures pour stabiliser le système, ce qui est dû à la grande inertie du béton. Mais même après ce temps la température de la surface extérieure de béton n'est descendue qu'à $17.1°C$. La partie la plus importante de la résistance thermique se trouve – comme voulu – dans la couche isolant.

Numériquement, toutes les configurations étudiées donnent la même solution, à une grandeur inférieure à la précision prêt. C'est le signe que le choix de la méthode des découpage n'affecte pas le résultat physique (!). Les différences entre les méthodes de résolution et entre les différentes configurations se manifestent dans le temps nécessaire et donc à la quantité de calculs nécessaire pour aboutir à la solution. La figure 8.6 montre une matrice contenant les temps de calcul[3] où dans les lignes nous comparons les différentes méthodes et dans les colonnes se trouvent les différentes configurations de découpage.

	L	**M**	**W**
GLOBAL	7.4	13.8	14.0
LOCAL	4.8	5.5	6.7
ASYNC	6.8	7.5	8.1

FIG. 8.6 - *Les temps de calculs pour les différentes configurations et méthodes de résolution.*

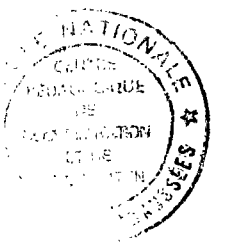
La comparaison entre les différentes configurations montre que le choix **L** est toujours le plus rapide. Dans cette configuration, nous ne faisons pas de calcul hiérarchisé. Tous les modules se trouvent au même niveau. Si un découpage existe, comme pour les cas **M** et **W**, c'est la configuration **M** qui est la plus rapide. Ceci nous conforte dans ce sens que le choix de découpage issu des considérations physique donne également des bons résultats d'efficacité numérique. Le choix de l'endroit du découpage n'est pas toujours évident, surtout quand il n'y a pas de distinction apparente entre les composants d'un système. Ce problème est approfondi dans le chapitre 9.

Entre les différentes méthodes employées dans cette expérimentation, on remarque également de grandes différences. La méthode LOCALE est bel et bien la plus rapide pour toutes les configurations. Ceci est le résultat d'une évaluation incomplète de la matrice globale de dépendances C. Pour déterminer la

3. C'est le temps *user* d'UNIX en secondes entre le pas zéro et la fin de la simulation (75h temps simulé) sur une machine monoprocesseur.

sensibilité sur la variation de la température de l'interface entre le coefficient de EXT et l'ISOLANT, nous n'évaluons pas les modules de BÉTON et INT.

La méthode ASYNC est essentiellement une méthode LOCALE avec en plus des tâches indépendantes pour les interfaces. Cela donne plus de liberté sur l'ordre de calcul, ce qui cause des itérations supplémentaires au niveau algorithmique. Comme nous exécutons le programme sur une machine monoprocesseur, il n'y a pas de vraie distribution de calcul. Le système d'exploitation doit donc changer encore plus souvent pour attribuer le processeur aux taches actives suite au nombre accru de tâches.



8.2 Cellule de Hambourg

Le nom de « *cellule de Hambourg*» désigne un problème bien défini de confort thermique, présenté pour la première fois à une conférence à Hambourg (d'où le nom) par le groupe RAMSES pour une démonstration de leur programme Zoom [10]. Les résultats issus de simulations de ce problème sont vérifiés à maintes reprises par d'autres logiciels comme Neptunix et Spark (voir §§ 2.3.3 et 2.3.4, [37] et [38]). C'est la raison pourlaquelle nous reprenons aussi le même exemple. Il sert également à vérifier la simulation par Motor-2. En même temps, nous pouvons comparer la facilité de manipulation des différents environnements.

8.2.1 Description du problème

La cellule de Hambourg consiste en une pièce d'habitation comme le montre la figure 8.7. La pièce est excitée par des sollicitations climatiques sur sa façade extérieure sud qui comprend une fenêtre et par la température de la façade nord qui se trouve à l'intérieur de l'immeuble. Un radiateur électrique peut chauffer le local. Il est piloté par un régulateur dont la sonde se trouve à la surface intérieure du mur nord.

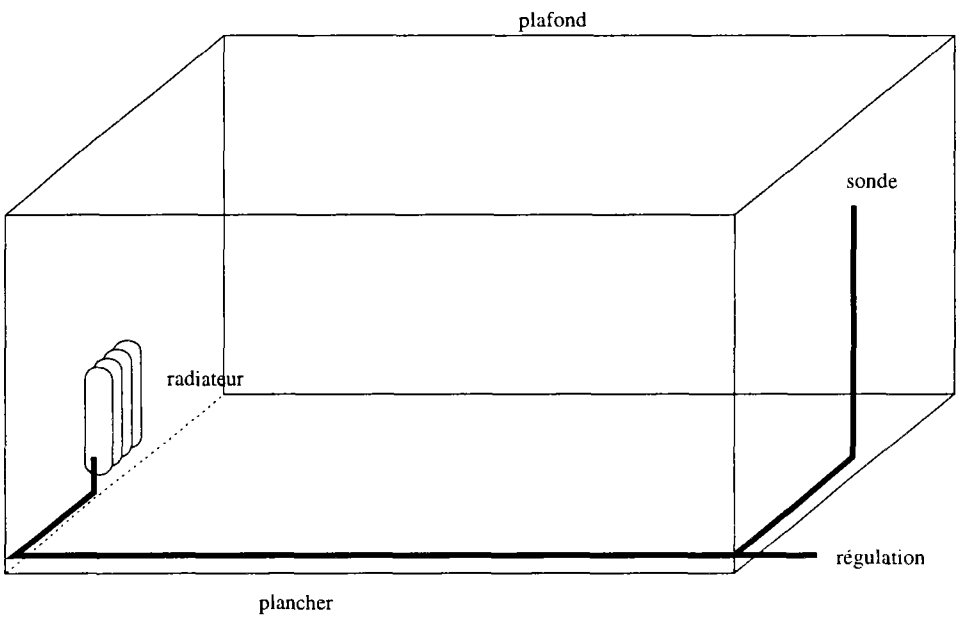


FIG. 8.7 - *La* cellule de Hambourg *vue au niveau technique.*

Passons maintenant au niveau physique de la description. Les phénomènes physiques et d'inévitables simplifications de la modélisation interviennent dans cette phase. L'intérieur de la pièce est divisé en trois parties entre lesquelles s'effectue un échange convectif forcé. Le radiateur chauffe l'air entrant et le pousse vers la partie haute. De là, l'air retourne vers la partie basse et entre de

nouveau dans le radiateur. Les deux zones d'air sont considérées comme respectivement parfaitement mélangées. Cette division en parties haute et basse a été projetée sur les murs nord et sud. Ces deux murs et la fenêtre sont modélisés par une conduction monodimensionnelle. Le plafond et le plancher de la pièce ainsi que les murs des cotés sont considérés comme adiabatiques, c'est à dire, ils ne subissent pas d'échange thermique. Le radiateur a également une capacité thermique, mais il insert une puissance de chauffage qui est déterminé par un système de régulation.

Nous spécifions encore quelques détails du problème aux niveaux mathématique et algorithmique. La conduction dans les murs et la fenêtre est traitée par une méthode aux différences finies. Pour faciliter la description de la partie haute du mur sud, les propriétés thermo-physiques de la fenêtre sont inclues par pondération à celles du mur qui la supporte. Un seul nœud capacitif est utilisé pour chacun des volumes de l'air et pour le radiateur. La régulation du chauffage fonctionne comme un thermostat linéarisé : la fonction analytique « *tangente hyperbolique* » est utilisée pour représenter le changement abrupt entre allumé – éteint.

$$P = \frac{P_{\max}}{2}\left(1 - \tanh\frac{T_s - T_c + T_b}{T_b}\right)$$

$P_{\max}$ est la puissance maximale du chauffage, T_s la température de la sonde, T_c la température de consigne, et T_b la largeur de bande de changement. On suppose une température constante à l'extérieur du mur nord créée par des thermostats parfaits. L'excitation climatique sur le mur sud est également supposée être une température constante pour des raisons de simplicité.

Dans la simulation avec Motor-2, nous reprenons également la même structure hiérarchique qui est déjà proposé dans la première publication originale [10]. Le problème global de la pièce est divisé en trois, le mur nord, l'intérieur de la pièce avec le système de régulation, et le mur sud avec la fenêtre. Chacune des parties géométriques est encore divisée en une partie haute et basse. Dans la figure 8.8 nous voyons le problème dans le schéma de la simulation numérique par Motor-2.

Nous retrouvons dans cette figure les modules qui sont utilisés dans la description du problème. Le module principal du problème est le module composé HH. Le schéma ne commence qu'avec ses trois sous-modules NORTH, ROOM et SOUTH. Le module SOUTH contient la partie basse du mur extérieur S2 et la fenêtre S1. Les deux modules sont du type FD1D1K pour la conduction monodimensionnelle. Des coefficients d'échange SOL_S1 et SOL_S2 les connectent avec la température constante de l'extérieur.

Le module composé de milieu contient l'essentiel du problème. Nous avons trois nœuds capacitifs, le volume d'air supérieure HO, le volume d'air inférieur BA, et la capacité calorifique du chauffage CE. Ces trois modules sont du type CN. Ils sont inter-connectés par le module central HTRN qui représente l'échange convectif entre les trois parties. À un débit constant, il y a transport circulaire du radiateur vers le volume en haut, puis vers le bas et rentrant de nouveau

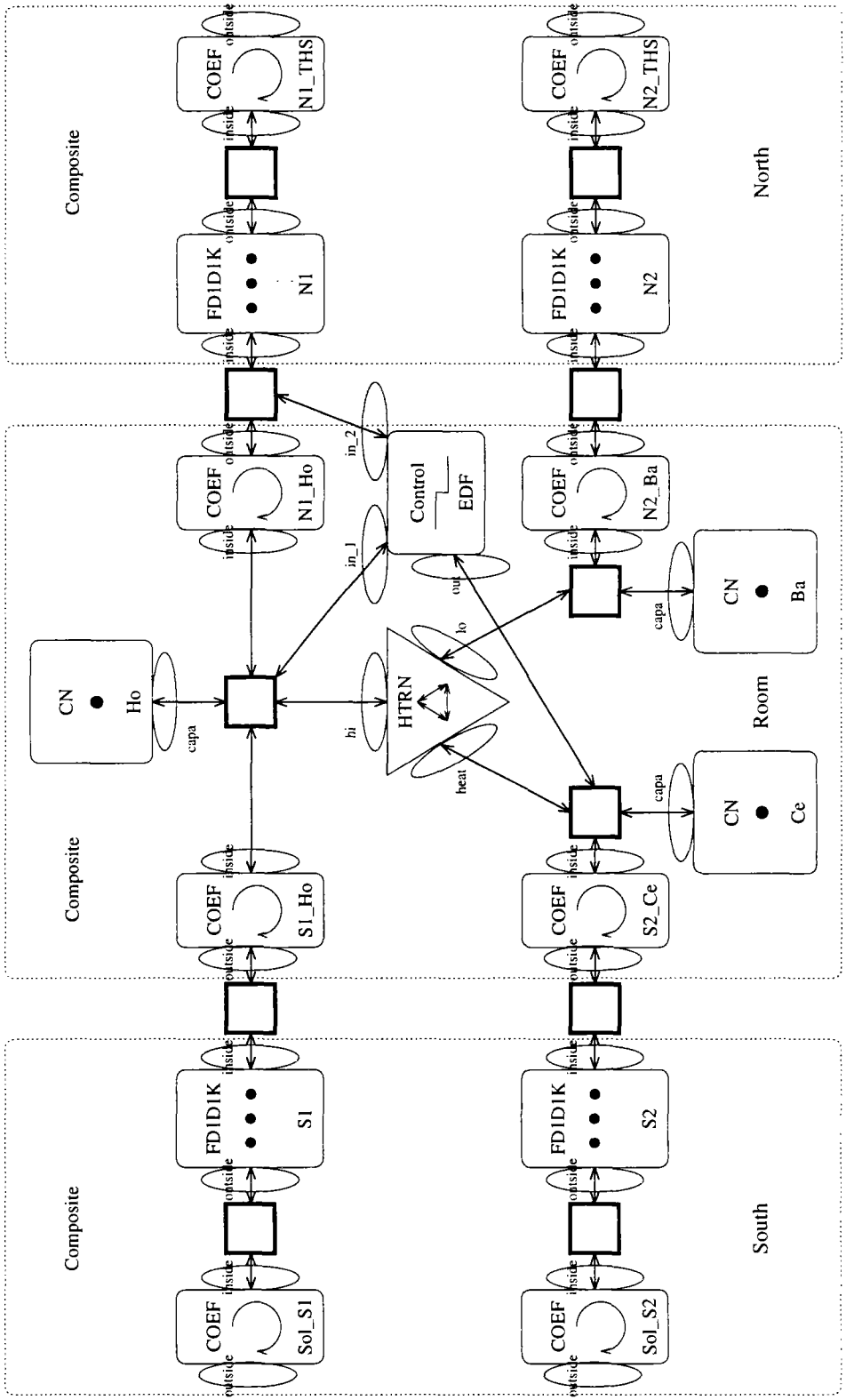


FIG. 8.8 - *La* cellule de Hambourg *vue au niveau algorithmique.*

dans le radiateur. Les trois modules sont également en échange convectif avec

les surfaces des murs sud et nord. Les modules S1_HO, N1_HO, S2_CE, et N2_BA représentent des résistances thermiques. La puissance de chauffage émise par le radiateur est déterminée par un dispositif de régulation. Le module EDF prend comme entrées les température du nœud HO et de la surface du mur N1. La moyenne des entrées est comparée avec la température consigne pour calculer le flux injecté dans le nœud CE.

Le troisième module composé du mur nord est symétrique de celui de la façade sud. Les modules N1 et N2 sont du types FD1D1K. Leur surface extérieure est connectée par des résistances thermiques N1_THS et N2_THS aux thermostats à températures constantes. Les températures constantes à l'extérieure comme à l'intérieure ne nécessitent pas de module dans Motor-2. Les valeurs initiales sont imposées dans un module, s'il n'y a pas de raccordement aux interfaces.

8.2.2 Résultats de la simulation et comparaison avec Spark

Nous avons utilisé des valeurs thermo-physiques qui ont déjà été utilisées dans une simulation avec Spark [38]. Pour simplifier les entrées, les modules de conduction monodimensionnelle ont tous les même valeurs. Elles se trouvent dans les tableaux 8.9, 8.10 et 8.11.

module	N1, N2, S1, S2	HO	BA	CE
capacité calorifique [J/K]	$3 \times 1{,}2\,10^5$	24.000	12.000	10.000
conductance [W/K]	300			

FIG. 8.9 - *Les valeurs thermo-physiques des murs et des volumes d'air.*

module	SOL_S1	SOL_S2	S1_HO	S2_CE
conductance [W/K]	36	18	24	6
module	N1_HO	N2_BA	N1_THS	N2_THS
conductance [W/K]	24	12	24	12

FIG. 8.10 - *Les coefficients d'échange.*

module	EDF
température consigne [K]	293
largeur de bande [K]	3
puissance maximale [W]	1000

FIG. 8.11 - *Les paramètres de la régulation.*

Les températures initiales sont de 287 K dans tous les modules. Nous déclenchons et suivons l'évolution des températures et des flux pendant 5000 secondes. Le régime permanent n'est pas encore atteint à ce moment, mais les données

disponibles de la simulation Spark s'y arrêtent. Comme pas de temps, nous avons choisi 5 secondes, car c'est le pas minimal que Neptunix détermine automatiquement. Les autres simulations faites par Spark et Zoom ont également utilisé cette valeur.

Le radiateur est donc mis en marche par la régulation afin d'atteindre la température de consigne. Il chauffe lui-même et l'air et introduit de cette façon une perturbation dynamique dans le système. La figure 8.12 montre l'évolution de la température du volume supérieur HO. Les deux autres figures 8.13 et 8.14 nous montrent l'erreur relative entre la simulation Motor-2 et Spark.

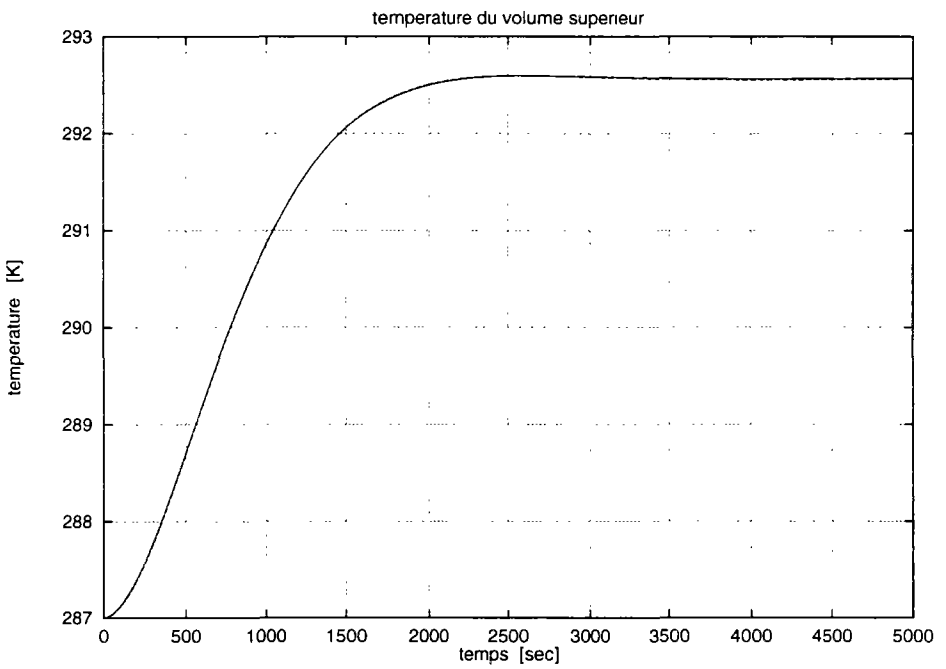


FIG. 8.12 - *L'évolution de la température* HO. *La ligne continue provient de Motor-2, la ligne pointillés de Spark.*

La température HO monte rapidement – pendant une demi-heure – et s'approche à quelques dixièmes de degrés près de la température de consigne. Environ trois quarts d'heure après le lancement, elle descend un peu et se stabilise ensuite à 292,6 K.

La superposition des résultats provenant de Motor-2 et Spark ne montre aucune différence visible (fig. 8.12). Les deux simulations donnent donc quasiment le même résultat. L'erreur relative, c'est à dire la différence entre Motor-2 et Spark normalisée est visible dans la figure 8.13. Elle est d'un ordre de 10^{-3} % ce qui est la précision demandée dans Motor-2. Le graphe de la figure 8.14 montre les résultats de Motor-2 par rapport aux résultats Spark. Cela donne une droite dans la diagonale. Ce type de graphe aide à trouver des incohérences entre deux simulation en cas de problème.

L'évolution du flux de chaleur qui est introduit dans la pièce par le système de chauffage est montrée dans la figure 8.15. Il part d'une puissance presque

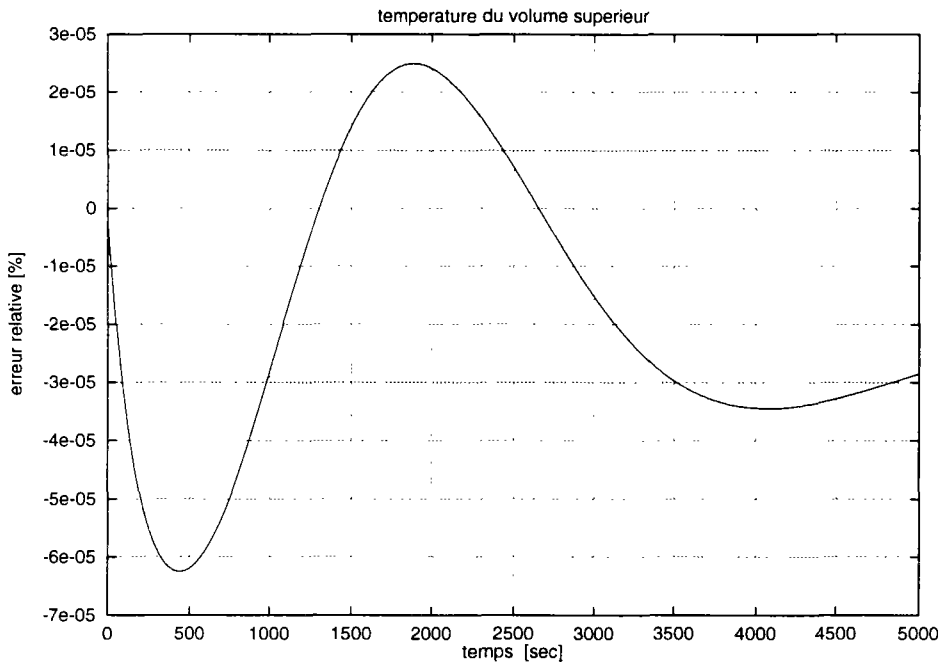


FIG. 8.13 - *L'erreur relative entre Motor-2 et Spark.*

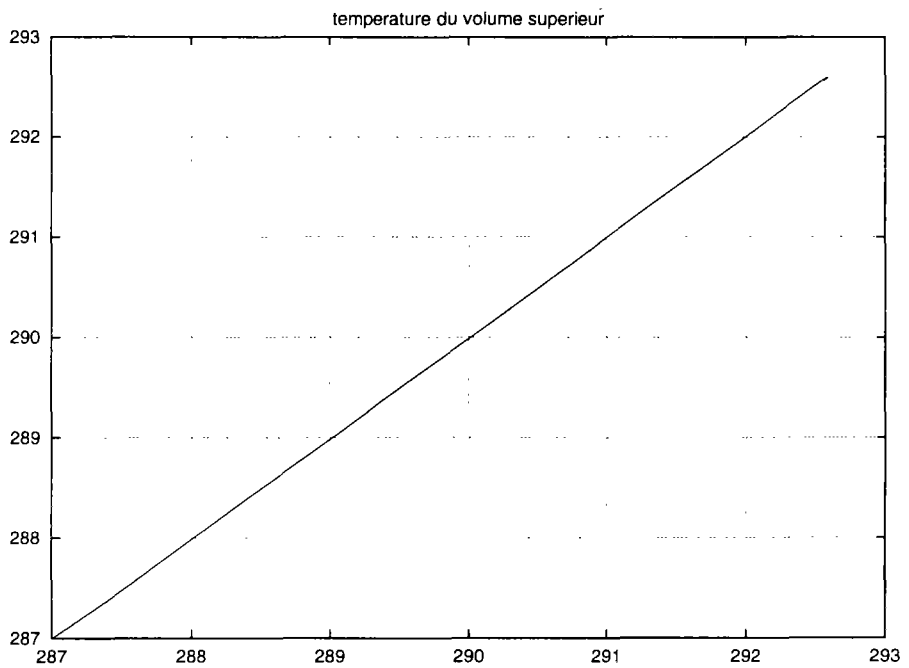


FIG. 8.14 - *Le rapport Spark sur Motor-2.*

maximale, et descend rapidement pendant la première demie heure et ensuite encore plus lentement. Au régime permanent (qui n'est pas montré dans cette figure), il arrive autour de 280 W.

L'erreur relative de cette variable dans la comparaison Spark – Motor-2 est

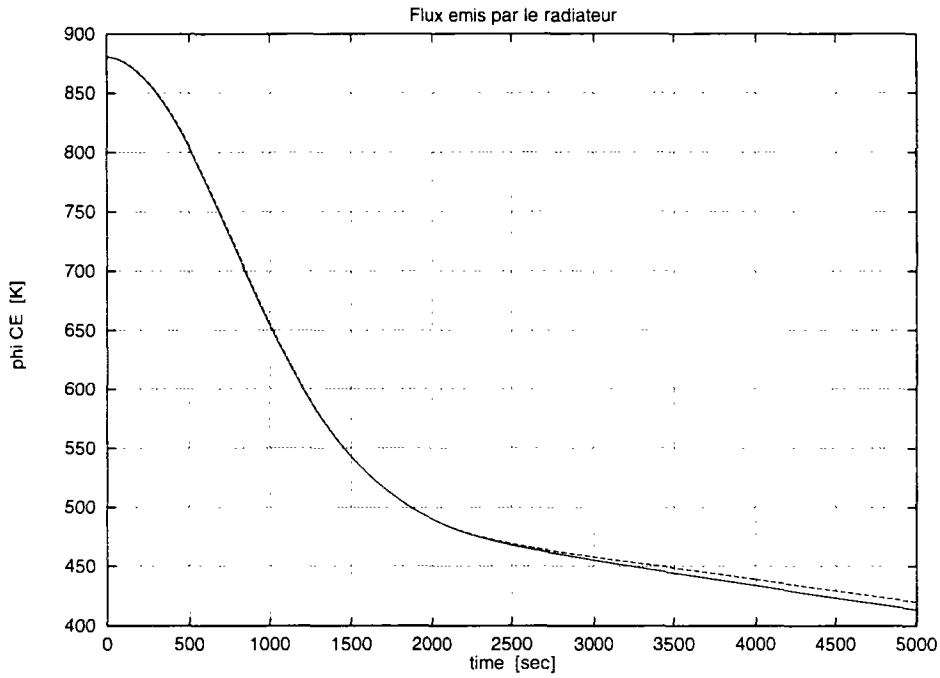


FIG. 8.15 - *Le flux introduit par le chauffage* EDF.

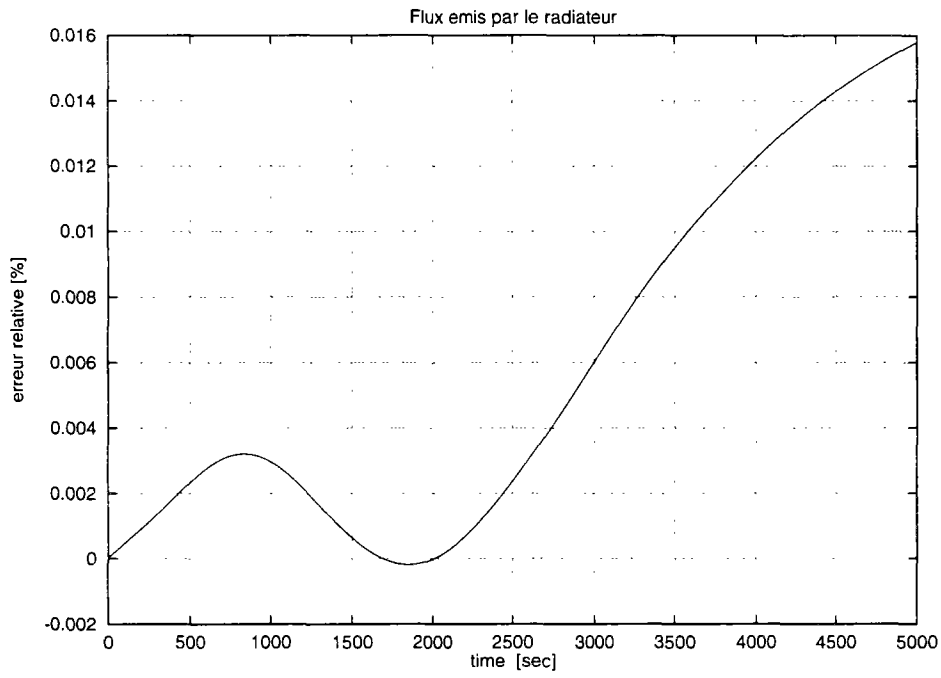


FIG. 8.16 - *L'erreur relative entre Spark et Motor-2 pour le flux de chauffage.*

un peu plus grande que pour la température du HO. Néanmoins elle reste largement dans les limites permises. D'autre variables ont été également suivies et comparées. Par exemple, les évolutions de la température du volume d'air inférieur BA et du flux de chaleur de cette partie dans le mur nord N2 se trouvent

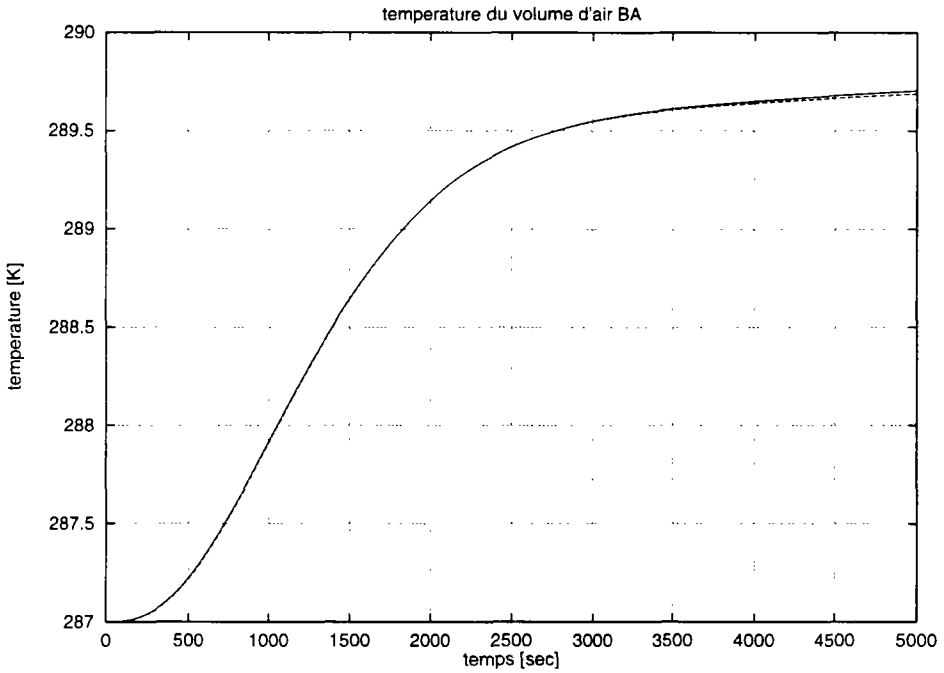


FIG. 8.17 - *L'évolution de la température du module* BA.

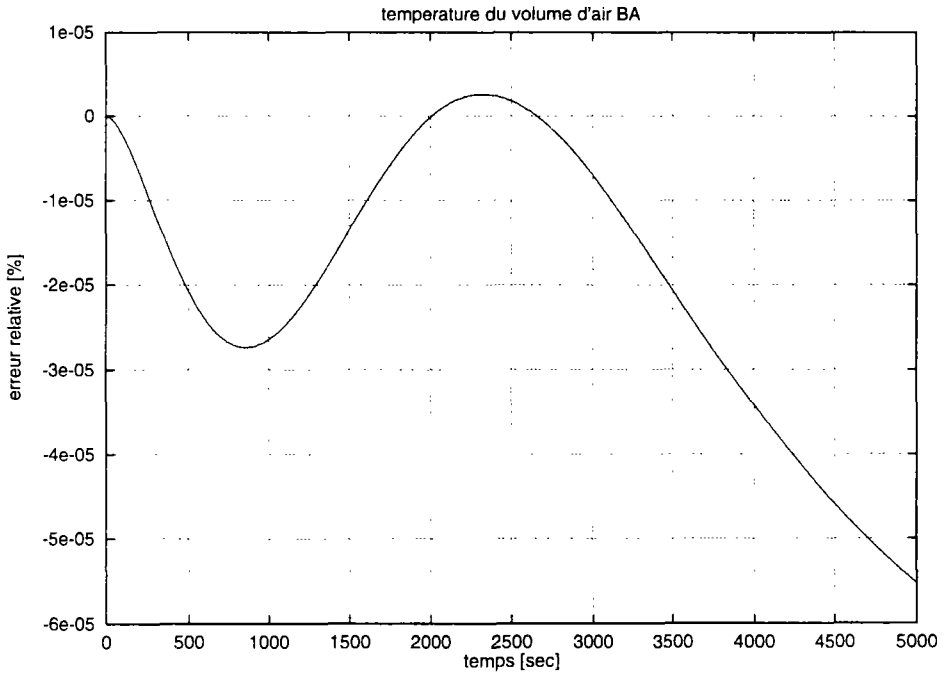


FIG. 8.18 - *L'erreur relative entre Spark et Motor-2 pour la température* BA.

dans les figures 8.17 et 8.19. Les figures 8.18 et 8.20 montrent les comparaisons avec les résultats correspondants de Spark.

Quelques tests simples ont été effectués pour comparer les méthodes internes de Motor-2. Pour cet exemple de la cellule de Hambourg, la méthode LOCALE

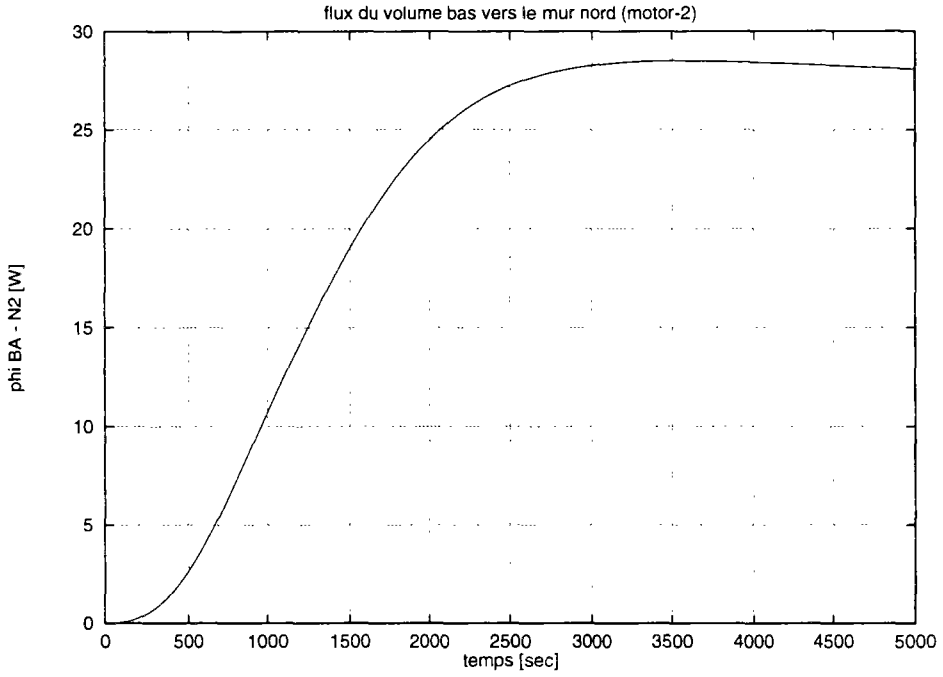


FIG. 8.19 - *L'évolution du flux de chaleur entre les module* BA *et* N2.

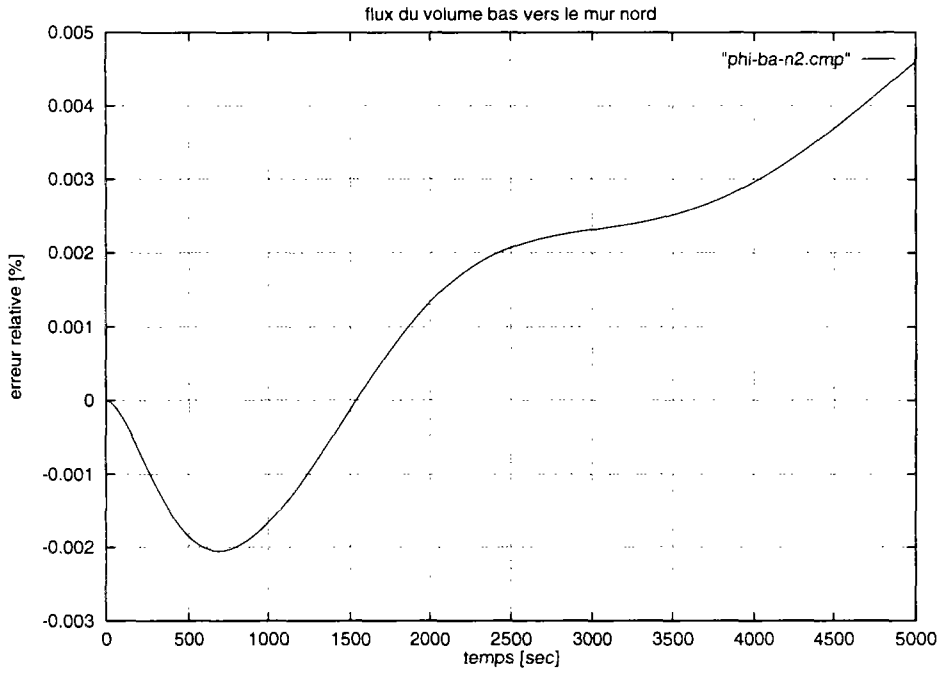


FIG. 8.20 - *L'erreur relative entre Spark et Motor-2 pour le flux entre* BA *et* N2.

de résolution indépendante par interface diminue de beaucoup le temps nécessaire de calcul. Dans la configuration décrite ici, la méthode GLOBALE met 192 secondes, la méthode LOCALE ne met que 55 secondes. C'est un facteur d'accélération de 3,5 (!).

8.2.3 Conclusion

La bonne correspondance entre les résultats obtenus par Motor-2 et et ceux obtenus par Spark, dans cet exemple, montre la fiabilité du programme Motor-2. L'erreur relative entre les deux système se mesure toujours en fractions de pourcent. La proximité des résultats est quand même surprenant compte tenu des approches très différentes de modélisation entre Spark et Motor-2. Il n'existe pas de calcul hiérarchisé dans Spark. toutes les équations qui constituent le problème sont écrites « à plat ». C'est ensuite l'algorithme interne qui connecte les équation et qui réduit le nombre de variables d'itération. Les modèles de Motor-2 contiennent directement les équations orientées. Les variables de sortie et de'entrée sont définies. L'utilisateur doit spécifier des raccordements entre les module par des interfaces.

Le même exemple a été également traité avec Zoom et Neptunix [37]. Les paramètres exactes de cette comparaisons ne pouvaient pas être retrouvés, mais les différences entre les résultats obtenus par Neptunix et Zoom se situent également dans le même ordre de grandeur.

8.3 Local incluant convection libre

8.3.1 Description du problème

Il s'agit de développer un modèle de simulation des mouvements d'air dans un local. Le principe de calcul repose sur un découpage du volume d'air de la pièce en plusieurs zones pour lesquelles on écrit les bilans de masse et d'énergie. L'objectif est de proposer un modèle 3D d'échange convectif destiné à la prédiction du confort thermique dans un bâtiment. Pour la modélisation de ce problème, nous avons repris l'approche de la publication [13].

Ce découpage consiste à subdiviser un local en zones parallèlépipédiques à génératrices verticales ou horizontales. Chaque zone est supposée à température homogène et à profil de pression hydrostatique. Les surfaces de contact entre ces zones sont supposées se comporter comme des parois fictives et totalement perméables. Les débits massiques entre deux zones obéissent à une loi de comportement en puissance découlant de la relation de BERNOULLI. Ils sont fonctions des pressions, des températures et de la taille des zones.

Le moteur de la convection provient de la température imposée sur les parois intérieures, le coefficient d'échange est supposé constant entre paroi et enceinte.

Dans les travaux de WURTZ, cette approche pour la convection est couplée avec d'autres phénomènes comme le rayonnement entre les surfaces de l'enceinte et la conduction dans les parois [34].

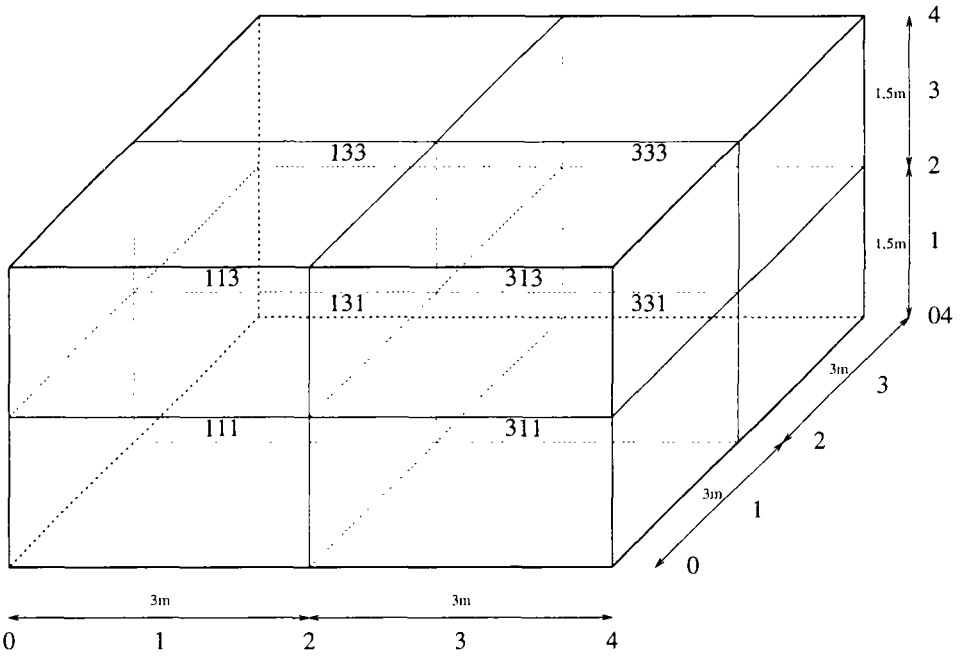


FIG. 8.21 - *Le schéma de la pièce.*

Nous considérons une pièce de $6 \times 6 \times 3m$ telle que présentée dans la figure 8.21. Un système de coordonnées compte de 0 à 4 dans les trois dimensions

qui servent d'indice de nœuds et de facettes. Les nœuds se trouvent toujours avec un indice impair, les facettes sur des indices pairs. Un nœud est donc localisé par ses trois coordonnées x, y et z. La cellule 131, par exemple, se trouve à l'indice 1 de x, 3 d'y, et 1 de z ; c'est le sous-volume gauche arrière bas.

8.3.2 Modélisation

Étant donné que les variations de pressions sont très faibles, on néglige la variable vitesse dans l'équation de BERNOULLI à l'intérieur des sous-volumes. Mais on écrit une équation de débit entre les deux sous-volumes en supposant une ouverture qui est la cause de la perte de charge Δp. Le type de représentation de mouvement d'air modélisé est donc celui de la figure 8.22.

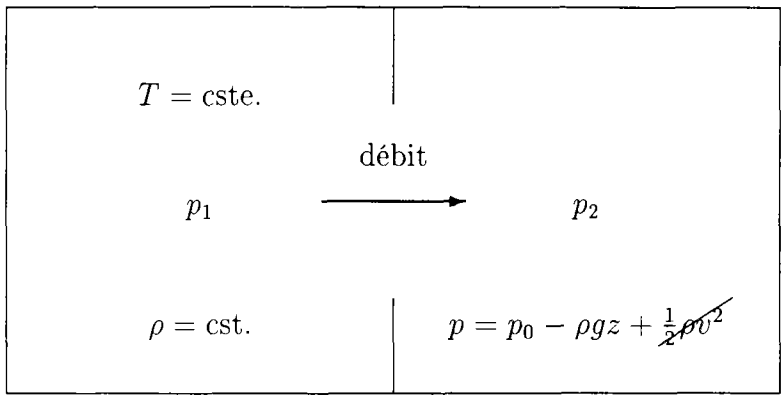


FIG. 8.22 - *Modélisation de la circulation de l'air.*

Au niveau des conditions aux limites, il s'agit de décrire à la fois les échanges d'air avec les parois et les « *fuites* » vers l'air extérieur dues aux bouches d'aération ou encore aux perméabilités de parois. Nous utilisons donc trois type de modèle :

1° les coefficients d'échange entre les parois et les nœuds de sous-volumes,

2° les équations décrivant les échanges et débits horizontaux,

3° les équations décrivant les échanges et débits verticaux.

Nous plaçons un module Motor-2 entre deux nœuds de sous-volume et entre les nœuds et les températures des parois (fig. 8.23). Comme les équation de circulation de l'air nécessitent toujours des valeurs des deux nœuds connectés, l'essentiel des calculs se trouve entre les nœuds.

Développement des équations

Un sous-volume a six sous-volumes voisins, désignés par *north*, *south*, *west*, *east*, *top*, et *bottom*. L'écart de pression entre deux sous-volumes entraîne un écoulement de vitesse v_m. Nous développons un ensemble d'équations simplifiées

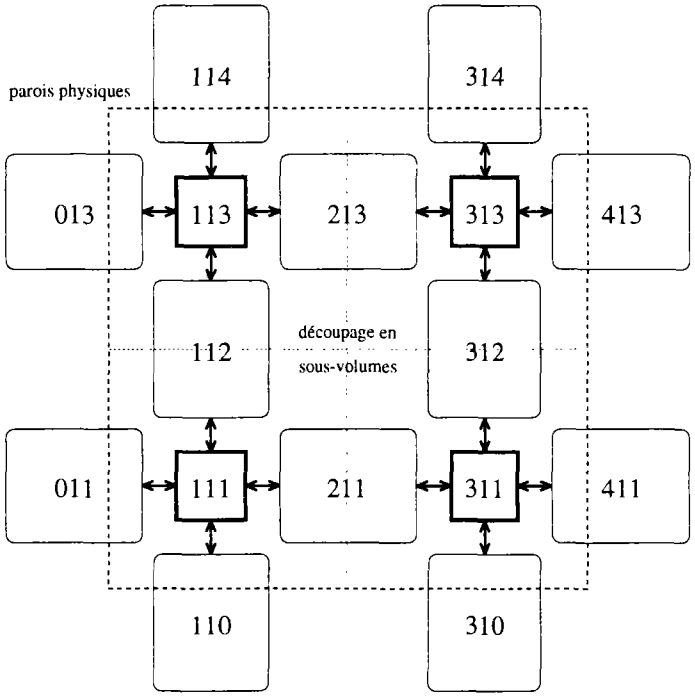


FIG. 8.23 - *Les modules de Motor-2 se trouvent entre deux nœuds. Ici sont montrés les modules et les interfaces de la tranche $y = 1$.*

basées sur l'équation de BERNOULLI. Elle permet d'écrire

$$p_1(z) = p_2(z) + \frac{1}{2}\rho_1 v_m^2 \tag{8.7}$$

d'où

$$\Delta p = \frac{1}{2}\rho_1 v_m^2 \tag{8.8}$$

ou encore

$$v_m = \sqrt{\frac{2}{\rho}}\Delta p^{0,5} \tag{8.9}$$

Or le débit massique théorique élémentaire vaut

$$\mathrm{d}\dot{m} = \rho v l \mathrm{d}z \tag{8.10}$$

nous obtenons

$$\mathrm{d}\dot{m} = \rho\sqrt{\frac{2}{\rho}}\Delta p^{0,5} l \mathrm{d}z \tag{8.11}$$

L'équation du débit est donc de la forme

$$\mathrm{d}\dot{m} = \rho k \Delta p^n l \mathrm{d}z \tag{8.12}$$

où k représente la perméabilité et n est choisi en fonction du type d'écoulement. Ce sont des coefficients empiriques recherchés expérimentalement.

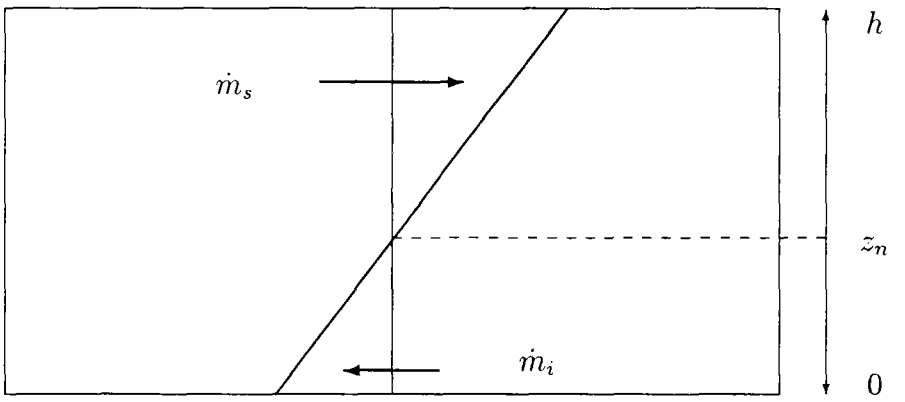


FIG. 8.24 - *Le profil des différences de pressions* Δp.

L'échange horizontal est décrit par deux flux. Ils proviennent d'un profil linéaire de différence de pression comme le montre la figure 8.24. On en déduit la cote z_n définie par $\Delta p(z_n) = 0$ et deux débits de sens opposés du part et d'autre de cet axe.

On a :

$$\Delta p(z_n) = 0 \tag{8.13}$$

d'où

$$z_n = \frac{p_1 - p_2}{(\rho_1 - \rho_2)g} \tag{8.14}$$

En intégrant $\dot{m}$ de z_n à h (la hauteur d'un sous-volume) on obtient :

$$\dot{m}_s = kl\rho(\Delta\rho g)^n \frac{(h - z_n)^{n+1}}{n+1} \tag{8.15}$$

puis en intégrant de 0 à z_n :

$$\dot{m}_i = kl\rho(\Delta\rho g)^n \frac{(z_n)^{n+1}}{n+1} \tag{8.16}$$

Aux débits de masse sont couplés des flux d'enthalpie. La puissance transportée entre deux sous-volumes s'écrit donc :

$$\phi = \dot{m}_s c_{p1} T_1 - \dot{m}_i c_{p2} T_2 \tag{8.17}$$

Le débit vertical entre deux sous-volumes s'écrit de la même manière comme :

$$\dot{m} = k\rho S(p - p_t)^n \tag{8.18}$$

Aux contacts des parois, le flux est transmis par échange convectif :

$$\phi = hS\Delta T \tag{8.19}$$

Équations des cellules

L'équation des gaz parfaits nous donne une relation entre p, ρ et T dans chaque sous-volume :

$$p = \rho RT \tag{8.20}$$

Il faut également vérifier les bilans des débits

$$\sum \dot{m} = 0 \tag{8.21}$$

et les bilans des flux

$$\sum \phi = 0 \tag{8.22}$$

dans tous les nœuds.

Modèles Motor-2

Nous avons donc construit deux nouveaux modèles dans la bibliothèque Motor-2, un pour la circulation d'air horizontale CONV_HORIZ et un pour la circulation d'air verticale. Les équations des bilans se prêtent bien pour les vérifications des interfaces dans Motor-2. Elles sont implémentées dans le nouveau modèle d'interface CONVECT_POINT. La figure 8.25 montre le schéma des équations dans Motor-2. Les interfaces sont placées directement sur les nœuds, les modules se trouvent toujours entre deux nœuds. Les interfaces envoient donc la température et la pression d'un sous-volume à tous les module d'échange. Ces derniers calculent les débits et les flux de chaleur avec lesquels ils répondent aux interfaces.

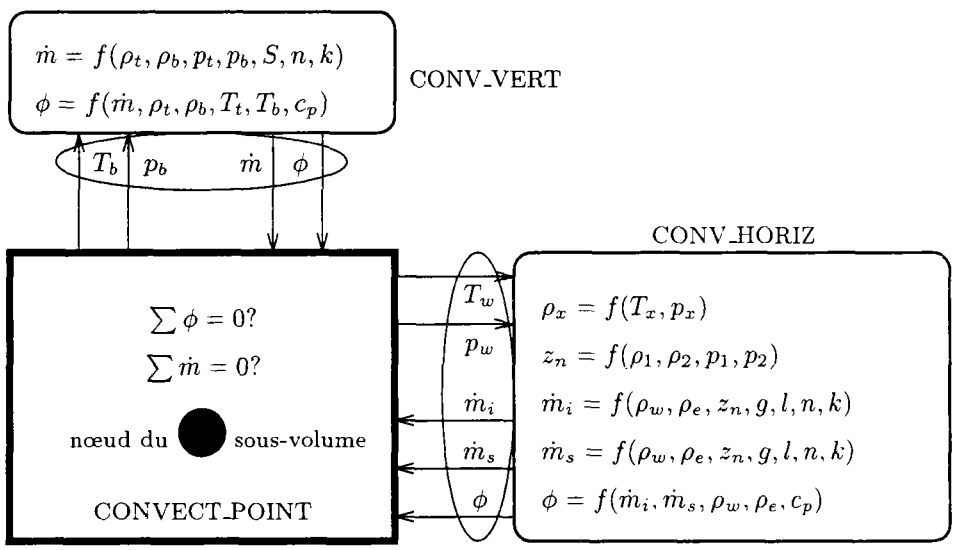


FIG. 8.25 - *Organisation des équations.*

Les modules prennent comme paramètres la géométrie pour déterminer la surface S et la hauteur h. Les coefficients qui spécifient le type d'écoulement et

la perméabilité sont n et k respectivement. Les autre grandeurs physique sont les constantes R et g.

Le découpage de la pièce en huit sous-volumes crée huit nœuds et donc huit interfaces internes. Chacun des nœuds est inter-connecté à ses voisins par six modules décrivant la circulation d'air ce qui nécessite 36 modules. À ce nombre s'ajoute les 24 modules d'échange convectif entre les surfaces des parois et les nœuds de sous-volumes. Le problème entier comprend donc 60 modules, ainsi que le module principal qui décrit les connexions.

8.3.3 Simulation et résultats

Étant donné le nombre important de variables et le fait que le système est fortement non-linéaire, obtenir la convergence est loin d'être trivial. Seule la méthode GLOBALE est capable de trouver une solution. L'algorithme interne utilisé est celui de Minpack, un paquetage dédié à la solution des systèmes d'équations non-linéaires. Un autre moyen de trouver la solution est le facteur d'amortissement qui ralentit la convergence, mais qui permet d'amortir les variations brusques dans le cas d'équations non-continues (voir D.1). C'est utile ici, car lors d'un changement du sens des débits, il faut également changer les variables température et pression. En effet, pour décrire un débit ou un flux, on prend uniquement en compte les variables correspondant à l'origine du mouvement de l'air.

Dans le cadre de cet exemple, nous avons testé une configuration simple dans laquelle on connaît *a priori* l'allure de la circulation de l'air. L'objectif principal consistait à montrer la faisabilité d'un tel type de modélisation et la capacité de Motor-2 à trouver une solution. C'est la raison pour laquelle nous ne faisons qu'une simulation statique. C'est à dire que les températures des parois restent constantes, et qu'un seul « pas de temps » est donc nécessaire.

Pour notre simulation, la paroi gauche est froide ($8°C$) ($x = 0$), toutes les autres étant adiabatiques à la température $18°C$. Les coefficients k de perméabilité sont mis à zéro pour les modules d'échange à travers les parois. Ils valent 1 à l'intérieur de la pièce. Le problème est symétrique au plan $y = 2$. Les quatre sous-volumes de devant devraient donner les mêmes résultats que les quatre sous-volumes de derrière. On ne devrait pas avoir d'échange entre les sous-volumes de devant et ceux de derrière.

Les figures 8.26 à 8.29 montrent les valeurs que nous avons obtenues dans une simulation avec Motor-2. Dans chacun des sous-volumes sont indiquées la température et la pression du nœud. L'épaisseur des flèches correspond à la « *vitesse débitante* » de convection[4] Nous voyons bien une circulation d'air en sens inverse des aiguilles d'une montre dans les figures 8.26 et 8.27.

Si l'on regarde les quatre sous-volumes de gauche ($x = 1$) de la figure 8.28, nous ne voyons que des débits verticaux qui descendent des sous-volumes supérieures. Par contre, dans les sous-volumes de droite ($x = 3$) de la figure 8.29,

4. C'est le débit divisé par la surface de la facette.

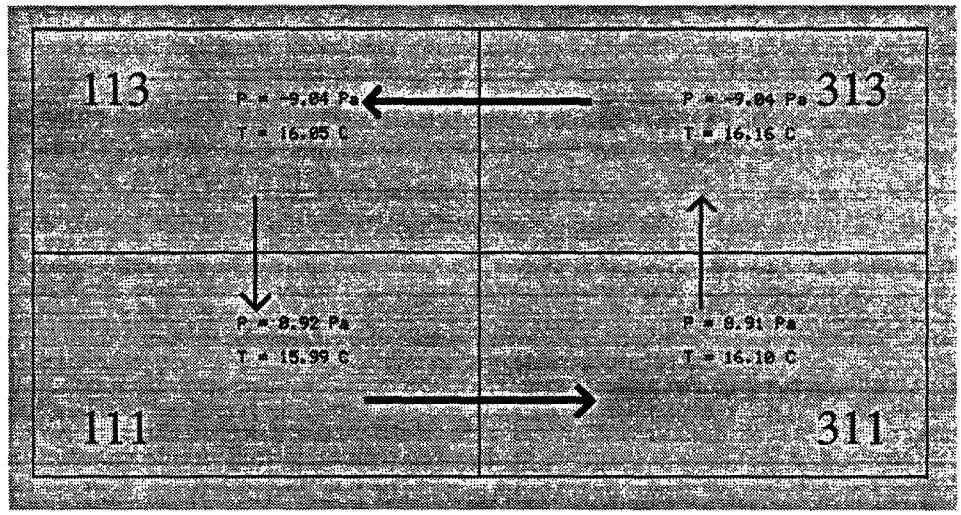


FIG. 8.26 - *La convection entre les quatre sous-volumes du premier plan ($y = 1$).*

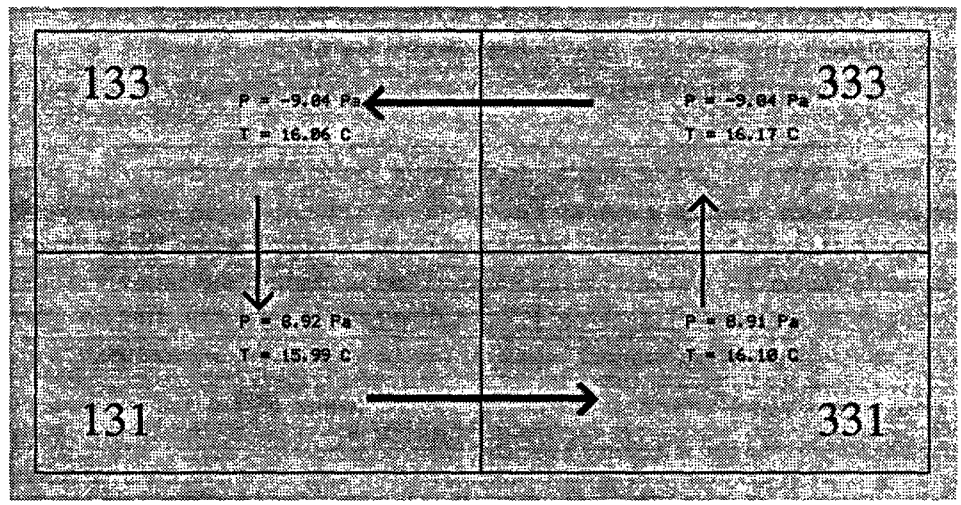


FIG. 8.27 - *La convection entre les quatre sous-volumes de l'arrière plan ($y = 3$).*

on remarque des échanges entre les sous-volumes de devant et celles de derrière. L'inspection des valeurs numériques montre que ces échanges sont très faibles. C'est certainement dû aux perturbations de l'algorithme de résolution. Au cours du développement de cet exemple, nous avons remarqué que le système est particulièrement sensible aux valeurs des paramètres et des limites. Un petit changement de valeurs d'entrée peut causer de graves problèmes numériques comme par exemple une divergence de calculs ou une multiplication des itérations nécessaires.

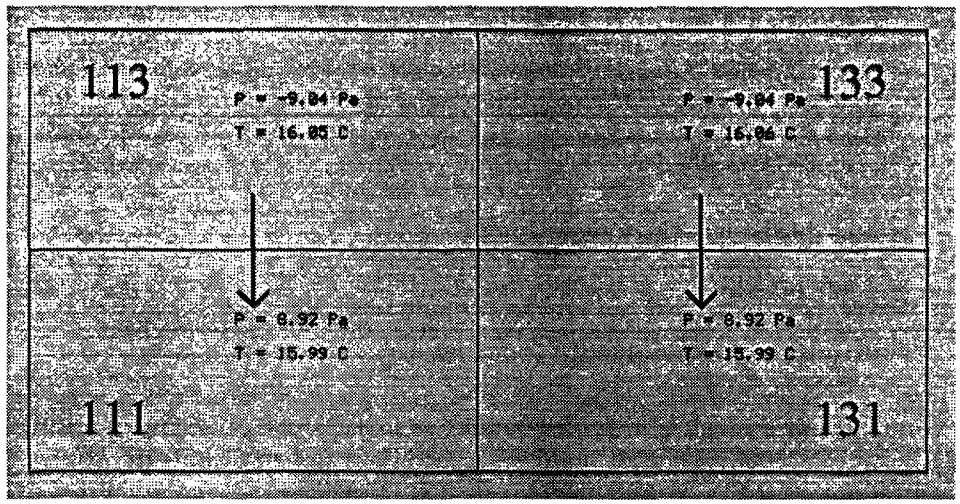


FIG. 8.28 - *La convection entre les quatre sous-volumes de gauche ($x = 1$).*

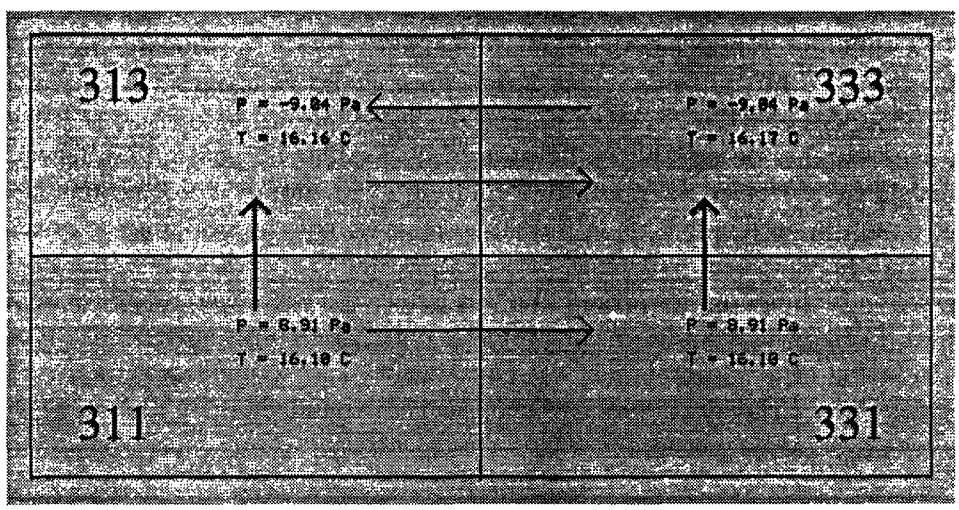


FIG. 8.29 - *La convection entre les quatre sous-volumes de droite ($x = 3$).*

8.3.4 Comparaison avec Spark

Le problème a été également simulé avec l'environnement Spark. Ici nous faisons aussi la distinction entre les calculs qui sont liés aux nœuds et calculs qui se trouvent entre deux nœuds. Dans des macros, on rassemble les équations correspondantes.

Pour chaque sous-volume, on écrit :

- le bilan des débits massiques,
- le bilan des flux de chaleur,

- une équation d'état,
- la variation hydrostatique

Pour chaque échange on calcule :

- le flux de chaleur entre deux zones,
- le débit massique,
- l'altitude de pression neutre (pour les échanges horizontaux)

On développe ensuite toutes les équations mises à plat, puis Spark réduit au maximum par substitution le nombre de variables pour faire les itérations sur un système réduit.

Les résultats obtenus avec Spark sont très proches des résultats de Motor-2. Les pressions et les débit sont les mêmes. Pour les températures, on trouve toujours un faible écart entre les deux systèmes.

La différence entre ces deux environnements réside surtout au niveau de la description du problème : dans Spark, on écrit les équations telle qu'on les a développées et c'est le noyau interne de Spark qui en déduit par réduction un petit nombre de variables d'itération. L'utilisateur ne peut qu'influencer le choix de ces variables retenues. La résolution et réduction des équations ne se font donc pas en fonction de la physique, mais en vue de minimiser le nombre de variables. Cela peut entraîner, par exemple, que l'algorithme travaille sur les masses volumiques qui ne varient que très peu ce qui pose des problèmes numériques. Dans la modélisation par Motor-2, l'utilisateur doit spécifier lui même l'orientation et le regroupement des équations et leur implémentation dans les modèles. Les variables d'itérations choisies correspondent donc davantage à la nature physique du phénomène.

8.3.5 Conclusion

L'ordre de grandeur des températures ainsi que de la circulation d'air correspond bien à la réalité attendue ce qui montre surtout la faisabilité de notre étude.

Dans cet exemple, nous couplons déjà deux phénomènes qui sont la circulation de l'air entre les sous-volumes et l'échange convectif entre les sous-volumes et les parois. Dans d'autres exemples, nous utilisons des modèle de rayonnement (cf. § 6.4.4) et des modèles des conduction en 1D et 2D (cf. exemples précédents et § 9.4). Une simulation qui prend en compte tous ces phénomènes semble maintenant possible. L'étude complète du comportement thermique d'un bâtiment par l'intermédiaire d'un modèle zonal couplé permettra d'une part d'homogénéiser la température dans un local ce qui réduira les consommations énergétiques et améliorera le confort, d'autre part de réduire les vitesses de l'air en particulier dans les locaux de grande hauteur.

Chapitre 9

Découpage

9.1 Présentation du problème

L'approche modulaire orientée objet est très intéressante pour l'utilisateur. Elle décrit le système à simuler en termes techniques et physiques. Les réflexions de la phase technique permettent de définir clairement des entités à la fois sur la hiérarchie et le découpage des modules. Ils sont habituellement des césures claires entre les composants, où des découpages paraissent naturels et faciles à faire (comme par exemple entre pompe, tuyau, capteur). L'impact du choix du découpage sur l'efficacité de la simulation n'est pas *a priori* connu. Le découpage qui est déduit du monde technique et physique, est-il numériquement efficace? Cette question est à la fois difficile et importante. On trouve une première discussion à ce sujet dans le cadre de l'environnement Motor-2 dans [61] ; nous la poursuivons ici.

La modularité de la description d'un système aboutit à un partitionnement du système global en sous-modules. Le chapitre précédant en donne quelques exemples. Posons le problème : quelle est la meilleure façon de découper un système pour que, non seulement la représentation reste la plus proche possible du système physique tel que nous le prenons, et pour que l'on atteigne la meilleure efficacité ? Bien entendu, l'efficacité ne dépend pas seulement du découpage, mais en première ligne des algorithmes de résolution employés. L'aspect « résolution» n'est pas traité ici ; nous nous penchons sur l'influence du découpage sur l'efficacité des calculs. Pour mieux comprendre les conséquences du découpage, trois types d'investigations ont été faites :

1° Quelques résultats peuvent être obtenus par des considérations théoriques dans le cas simple d'une série de résistances (mur multicouche 1D en régime permanent).

2° Pour le cas d'une plaque 2D, nous avons effectué quelques expérimentations avec Motor-2. Les bords de la plaque sont soumis à des températures différentes et nous avons mesuré le temps nécessaire dans le cas de différents découpages pour déterminer la distribution des températures sur la plaque.

3° Dans un troisième test, nous avons mesuré l'influence des profondeurs de hiérarchisation sur le temps de calcul. L'exemple est une simulation dynamique d'un local sollicité par un rayonnement solaire.

Nous considérons la simulation la plus efficace, celle qui est la plus rapide et qui consomme le moins de ressources possible. Dans les réflexions théoriques, nous voulons minimiser le nombre d'itérations ou accélérer la vitesse de convergence. Pour les expérimentations numériques, nous avons mesuré le temps de calcul nécessaire pour effectuer la simulation.

9.2 Considérations théoriques

9.2.1 Problème

Considérons une série de n résistances thermiques de valeur $1/h_i = r_i$. Entre les résistances se trouvent les températures T_i avec T_0 et T_n aux bords. Une résistance r_i est placée entre les températures T_i et T_{i+1} (voir la figure 9.1).

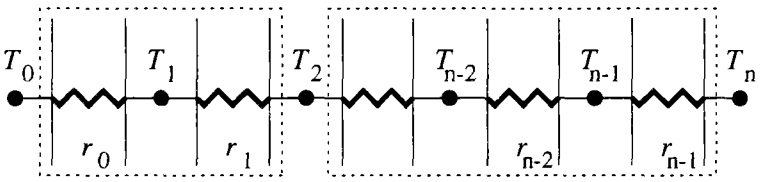


FIG. 9.1 - *Une série de résistances thermiques avec des températures imposées T_0 et T_n. Les cases pointillées indiquent un partitionnement binaire possible.*

Le flux de chaleur qui traverse un nœud s'écrit :

$$\phi_i = \frac{1}{r_{i-1}}(T_{i-1} - T_i) + \frac{1}{r_i}(T_i - T_{i+1}) = 0$$

Cette expression correspond à un problème de conduction thermique à travers une paroi multicouche avec des températures imposées sur les deux côtés (condition aux limites de type DIRICHLET). Le problème s'écrit sous la forme matricielle

$$\begin{bmatrix} -\frac{1}{r_0} - \frac{1}{r_1} & \frac{1}{r_1} & 0 & 0 & 0 \\ \frac{1}{r_1} & -\frac{1}{r_1} - \frac{1}{r_2} & \ddots & 0 & 0 \\ 0 & \frac{1}{r_2} & \ddots & \frac{1}{r_{n-3}} & 0 \\ 0 & 0 & \ddots & -\frac{1}{r_{n-3}} - \frac{1}{r_{n-2}} & \frac{1}{r_{n-2}} \\ 0 & 0 & 0 & \frac{1}{r_{n-2}} & -\frac{1}{r_{n-2}} - \frac{1}{r_{n-1}} \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_{n-2} \\ T_{n-1} \end{bmatrix} = \begin{bmatrix} -\frac{1}{r_0}T_0 \\ 0 \\ \vdots \\ 0 \\ -\frac{1}{r_{n-1}}T_n \end{bmatrix}$$

La solution classique pour ce système d'équations est pour une température T_i au milieu, $1 \leq i \leq n-1$

$$T_i = \frac{\frac{T_0}{\sum_{p=0}^{i-1} r_p} + \frac{T_n}{\sum_{p=i}^{n-1} r_p}}{\frac{1}{\sum_{p=0}^{i-1} r_p} + \frac{1}{\sum_{p=i}^{n-1} r_p}} \tag{9.1}$$

Le problème que l'on se pose maintenant est comment trouver un partitionnement *optimal* pour que la convergence d'un algorithme de type NEWTON-RAPHSON se fasse le plus rapidement possible. Au niveau physique, on peut interpréter le partitionnement comme la division de la paroi en plusieurs ensembles de couches. Ces ensembles sont des *boîtes noires* avec leurs propres températures et flux thermiques et ses algorithmes internes pour les déterminer (cf. chap. 6). Un partitionnement est considéré comme *optimal* quand la convergence est la plus rapide de celles que l'on obtient avec tous les partitionnements possibles au niveau supérieur. Ce n'est pas le coût global en calculs, car nous ne faisons aucune hypothèse sur les algorithmes internes des sous-blocs. Le coût en calcul d'un module composé est la somme du coût correspondant aux sous-modules plus le coût de résolution des couplages.

9.2.2 Partitionnement binaire

Nous ne considérons que les cas où les résistances ou les blocs de résistances sont groupés par paires. C'est ce que nous appelons un partitionnement binaire. Par conséquent, la vitesse de convergence au niveau supérieur (entre les deux blocs) est simple à déterminer. Si le partitionnement est fait à la température T_i entre la résistance r_i et r_{i+1}, nous obtenons la valeur de T_i, s'il y a itération entre deux blocs, par la condition de flux (expression de l'interface)

$$T_i = \frac{1/r_{i-1}T_{i-1} + 1/r_i T_{i+1}}{1/r_{i-1} + 1/r_i} \tag{9.2}$$

Maintenant, T_{i-1} et T_{i+1} peuvent être obtenus par l'équation 9.1 (l'expression des sous-modules). Elles sont donc calculées par leurs blocs (avec une méthode numérique quelconque) et avec les températures aux limites T_0 et T_i pour T_{i-1}, T_i et T_n pour T_{i+1}. Ces températures internes dans les blocs se calculent donc comme

$$T_{i-1} = \frac{\frac{T_0}{\sum_{p=0}^{i-2} r_p} + \frac{T_i}{\sum_{p=i-1}^{i-1} r_p}}{\frac{1}{\sum_{p=0}^{i-2} r_p} + \frac{1}{\sum_{p=i-1}^{i-1} r_p}}$$

et

$$T_{i+1} = \frac{\frac{T_i}{\sum_{p=i}^{i} r_p} + \frac{T_n}{\sum_{p=i+1}^{n-1} r_p}}{\frac{1}{\sum_{p=i}^{i} r_p} + \frac{1}{\sum_{p=i+1}^{n-1} r_p}}$$

En combinant la condition de flux (éq. 9.2) et les deux solutions des blocs pour les températures T_{i-1} et T_{i+1} autour du point de partitionnement, nous obtenons une équation qui exprime la température à la césure T_i en fonction d'elle-même $T_i = f(T_i, T_0, T_n)$. Ceci est la base de la méthode itérative pour calculer T_i. On peut ensuite exprimer le rapport de convergence comme

$$\frac{1}{1/r_{i-1} + 1/r_i} \left[\frac{1/r_{i-1}}{1 + \frac{r_{i-1}}{\sum_{j=0}^{i-2} r_j}} + \frac{1/r_i}{1 + \frac{r_i}{\sum_{j=i-1}^{n-1} r_j}} \right] \tag{9.3}$$

Cette expression 9.3 est positive et inférieure à 1, si tous les coefficients de transfert h sont positifs. Pour accélérer la convergence, il faut minimiser cette expression en choisissant un endroit de découpage i. Minimiser l'expression 9.3 revient à maximiser l'expression suivante

$$\frac{\frac{1}{\sum_{j=0}^{i-1} r_j} + \frac{1}{\sum_{j=i}^{n-1} r_j}}{\frac{1}{r_{i-1}} + \frac{1}{r_i}} \tag{9.4}$$

Pour un ensemble de résistances données r_j, l'expression 9.4 doit être maximale. Pour mieux comprendre cette expression, regardons les deux exemples d'une isolation entre deux blocs conductifs et d'un élément conducteur entre deux blocs de résistances.

Si toutes les résistances r_j sont égales et ont la même valeur r sauf une r_k dont la valeur R est beaucoup plus grande que r, alors le découpage doit être fait au nœud T_{k+1} si r_k se trouve dans la deuxième moitié ($k > n-1-k$) et au nœud T_k pour r_k dans la première moitié ($k < n-1-k$). Ceci veut dire que la meilleure césure doit être faite au ras de l'isolant du coté la plus proche du bord (voir fig. 9.2). Ce résultat soutient donc l'avis qu'une isolation sépare deux blocs et un découpage à cet endroit parait naturel.

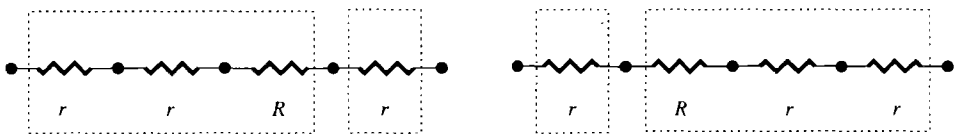


FIG. 9.2 - *Le découpage binaire se fait à la résistance isolante du coté la plus proche du bord.*

Si toutes les résistances r_j sont égales et ont la même valeur r sauf une r_k dont la valeur est beaucoup plus petite que r, alors le découpage optimal se trouve aux nœuds T_1 ou T_{n-1}, c'est à dire aux bords de la chaîne de résistances. Les deux découpages (T_1 ou T_{n-1}) donnent la même vitesse de convergence. Si $k = 0$ ou $k = n-1$, le découpage doit être au bout opposé de la petite résistance.

Nous pouvons conclure de ces deux exemples que le partitionnement binaire doit se trouver proche des grandes résistances et loin des petites résistances. Une règle simple résultante peut être le raisonnement suivant : une grande résistance entraîne un petit flux, nous avons donc un couplage faible et nous pouvons couper ici. Bien entendu, ces résultats ne sont valables aussi directement que pour des systèmes non-capacitifs.

9.3 Expérimentation : le cas 2D

Le cas 2D ne se laisse pas analyser aussi facilement que le cas monodimensionnel. Nous l'avons expérimenté numériquement sur quelques cas. Le problème étudié est une plaque composée de 8×8 éléments comme dans la figure 9.3.

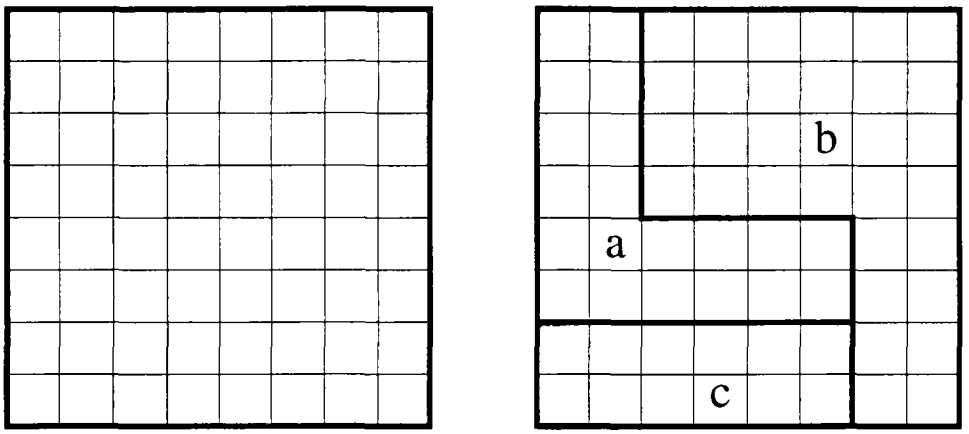


FIG. 9.3 - *Une surface carrée avec un maillage de 8 × 8 éléments. À droite se trouve un exemple d'un partitionnement en trois sous-modules.*

L'équation interne des éléments est du type différences finies sans capacité interne dans les nœuds (plaque non-capacitive).

Nous générons des partitionnements de plaque par un logiciel écrit en Prolog qui est dérivé des travaux de ALLAZ [2]. Ce programme prend comme paramètre le nombre de sous-modules souhaités et génère l'ensemble de tous les partitionnements possibles. Par quelques étapes intermédiaires, nous créons la description en format SYMBOL pour Motor-2. Ceci comprend les fichiers nécessaires des cellules élémentaires, les modules composés avec les interfaces internes pour le raccordement entre les mailles et le module composé principal qui raccorde les partitions.

Ensuite la simulation par Motor-2 est lancée. Nous déterminons le coût des calculs comme étant le temps nécessaire pour que Motor-2 calcule l'état stationnaire. Comme il n'y a pas de capacité dans les éléments, cela veut dire que l'on calcule un pas de temps. Pour chaque configuration, nous stockons plusieurs informations comme le temps de simulation, la longueur des frontières internes, le nombre de partitions, le nombre d'éléments par partitions, etc. Suite aux variations inévitables de temps mis pour des configurations identiques à cause des charges différentes des ordinateurs, les mêmes cas sont simulés plusieurs fois afin d'obtenir des meilleurs estimations sur le coût de calcul.

Une première étude qualitative se fait en regardant simplement le fichier créé. Des résultats plus quantitatifs peuvent être obtenus par un post-processeur qui crée une table des temps nécessaires et qui calcule la multi-regression linéaire de toutes ces données. Ce post-processeur est écrit un Maple et utilise ses fonctionnalités de calcul formel. Finalement on obtient une corrélation entre le temps de calcul et des puissances de la surface des partitions et de la longueur interne des frontières jusqu'à l'ordre 3. Nous mesurons également les différences entre les valeurs initiales et les solutions pour toutes les cellules.

Plusieurs configurations ont été testées. La plaque a été coupée en deux, trois ou quatre partitions. Plusieurs centaines de formes différentes pour chaque nombre de partition ont été étudiées. Différentes températures aux bords mon-

trent une influence des conditions aux limites sur les coûts de calcul. Mais on constate aussi que la distribution des conductivités sur la plaque et leurs changements par rapport aux partitions influencent le temps nécessaire.

L'interprétation des résultats que nous avons obtenus n'est pas évidente. Nous avons trouvé quelques configurations dont les résultats semblent à première vue contradictoires ou tout au moins difficiles à expliquer.

9.3.1 Cas homogène

Les tests suivants ont été exécutés sur une plaque homogène de conductivité $0.001 W/m^2K$. Les températures de bord étaient fixées par une condition de DIRICHLET à 5°C en bas, 10°C en haut, 15°C à gauche, et 20°C à droite. L'état initial est de 19°C partout. La figure 9.4 montre la solution, c'est à dire le régime permanent, du problème. On voit bien les températures fixées aux bords. Le champ de températures à l'intérieur de la plaque crée une forme de selle. Le « haut » de plaque se trouve derrière, le « bas » au premier plan de la figure.

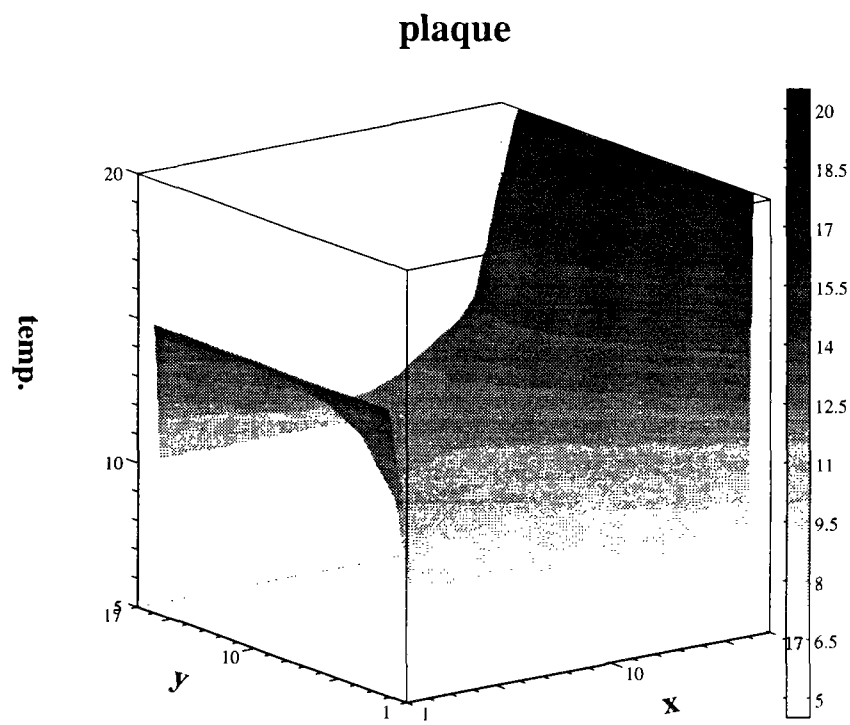


FIG. 9.4 - *Le champ de températures sur la plaque avec des températures imposées aux bords de 5° C en bas, 10° C en haut, 15° C à gauche, et 20° C à droite.*

Qualitativement on peut remarquer que le meilleur découpage est celui où les partitions sont de tailles comparables. L'introduction d'une très petite partition dans une configuration aboutit à une simulation plus longue. Ce résultat paraît

naturel, car une nouvelle petite partition ne change que très peu les partitions existantes, mais elle accroît la longueur des frontières et donc la taille du système d'équations qui est à résoudre (inversion de la *Jacobienne*).

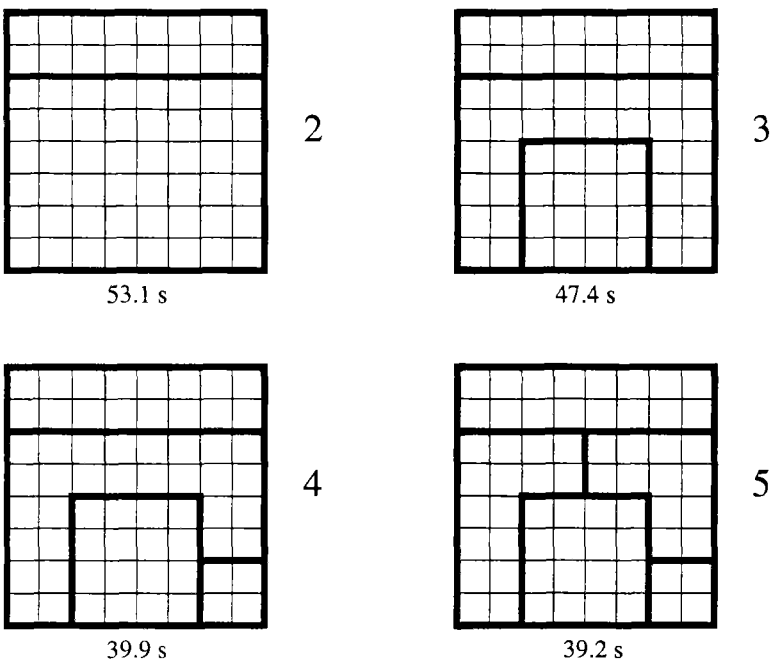


FIG. 9.5 - *Exemple des partitionnements où l'ajout d'une partition supplémentaire accélère les calculs.*

Par ailleurs, si un partitionnement existe déjà, par exemple imposé par la description physique, il peut être parfois favorable d'introduire des partitions supplémentaires. En général, si on a un nombre faible de partitions, on accélère la simulation en ajoutant d'autres partitions (d'une taille comparable). La figure 9.5 montre des temps de calcul décroissant pour un découpage en deux, trois, quatre et cinq partitions. Il faut noter ici, qu'il n'est pas évident de trouver un tel exemple. On n'est pas vraiment certain de diminuer à coup sûr le temps de calcul en ajoutant une partition.

Cette conclusion qualitative est confirmée par des corrélations. Elles sont calculées par le post-processeur sur un ensemble de partitionnements de la plaque. Regardons premièrement les corrélations qui sont indépendantes de la différence entre solution et valeur initiale. Nous désignons par C_i les coûts de calcul (le temps nécessaire) pour un partitionnement donné; C_2 désigne donc les coûts de calcul pour un partitionnement binaire. Les σ_i sont les sommes normées des i^{es} puissances des surfaces des sous-modules, et l est la longueur normée des frontières internes. Les normes sont établies par rapport à la surface globale (64) et par rapport à la longueur interne globale ($2 \times 7 \times 8 = 112$). La corrélation entre la formule construite et les durées des simulations est de 0.79 pour le partitionnement binaire, 0.69 pour le partitionnement tertiaire, et 0.51

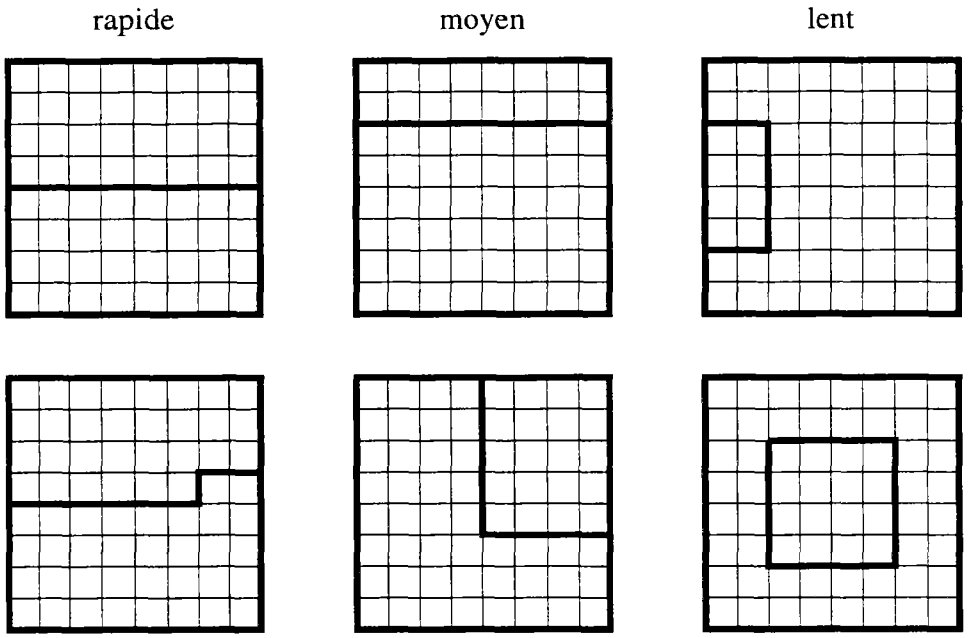


FIG. 9.6 - *Exemple des partitionnements en deux avec un temps de calcul croissant.*

pour le partitionnement quatuaire.

2 partitions corrélation = 0.79 (cf. fig. 9.6).

$$\begin{aligned} C_2 = {} & 219.5 - \\ & 1093.3\sigma_1 + 2300.82\sigma_2 - 1095.9\sigma_3 + \\ & 0.4l - 98.3l^2 - 34.1l^3 \end{aligned}$$

3 partitions corrélation = 0.69 (cf. fig. 9.7).

$$\begin{aligned} C_3 = {} & 215.3 - \\ & 130.1\sigma_1 - 1775.3\sigma_2 + 4163.8\sigma_3 + \\ & 109.6l - 33.1l^2 - 6.4l^3 \end{aligned}$$

4 partitions corrélation = 0.51 (cf. fig. 9.8).

$$\begin{aligned} C_4 = {} & 268.3 + \\ & 123.4\sigma_1 - 562.4\sigma_2 + 819.3\sigma_3 + \\ & 145.8l - 222.4l^2 + 57.7l^3 \end{aligned}$$

Les résultats et les corrélations dépendent aussi des conditions aux limites. C'est surtout le cas, si le partitionnement suit des isothermes de la solution. Ici, le même partitionnement peut donner des résultats très différents, si les conditions aux limites ne correspondent pas aux frontières des partitions. Cela peut

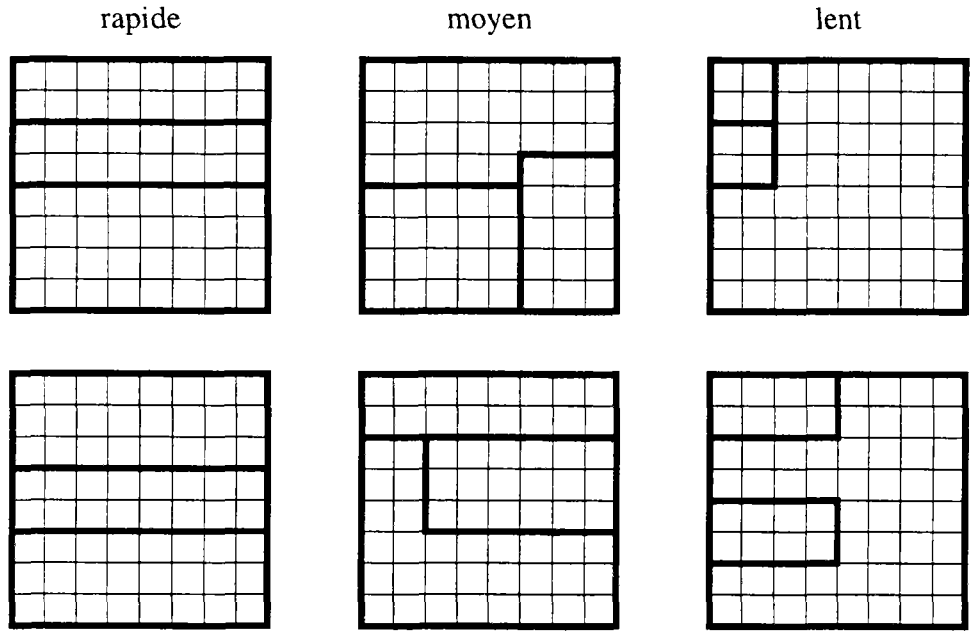


FIG. 9.7 - *Exemple des partitionnements en trois avec un temps de calcul croissant.*

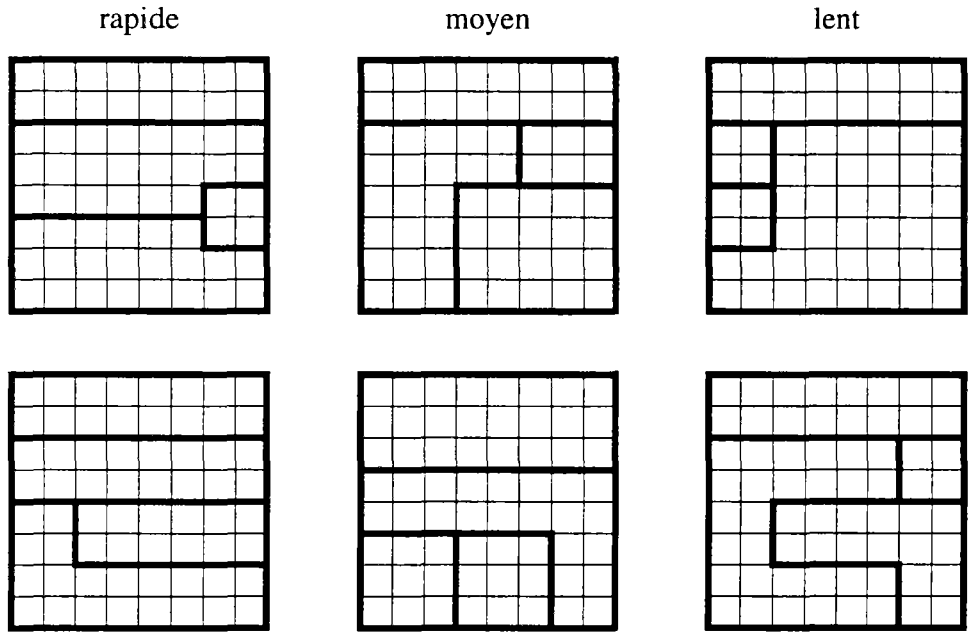


FIG. 9.8 - *Exemple des partitionnements en quatre avec un temps de calcul croissant.*

s'expliquer par le fait que la valeur initiale est plus éloignée de la solution, et il faut donc plus de temps pour converger. En introduisant cette différence dans les corrélations, nous avons défini un paramètre supplémentaire ξ qui contient la somme des différences entre les valeurs initiales et les solutions le long des frontières.

$$\xi_i = \int_{\text{frontières}} (T_{\text{init}}(s) - T_{\text{sol}}(s))^i \mathrm{d}s$$

où T_{init} est la valeur initiale, T_{sol} la valeur de la solution, s suit les frontières internes, et i un entier $(1, 2, 3)$. Les corrélations obtenues sur les résultats ne changent que peu. Elles valent $0.86, 0.71, 0.52$ pour les formules suivantes.

2 partitions corrélation $= 0.86$.

$$\begin{aligned} C &= 408.1 - \\ &\quad 806.9\sigma_1 + 1593.9\sigma_2 - 756.6\sigma_3 + \\ &\quad 1262.8\xi_1 + 3306.5\xi_2 + 2966.7\xi_3 - \\ &\quad 78.1l + 108.7l^2 - 24.4l^3 \end{aligned}$$

3 partitions corrélation $= 0.71$.

$$\begin{aligned} C &= -213.3 + \\ &\quad 49.8\sigma_1 - 2448.6\sigma_2 + 4856.0\sigma_3 - \\ &\quad 3680.1\xi_1 - 14984.3\xi_2 - 16699.0\xi_3 + \\ &\quad 495.8l - 256.4l^2 + 35.1l^3 \end{aligned}$$

4 partitions corrélation $= 0.52$.

$$\begin{aligned} C &= 207.6 + \\ &\quad 116.1\sigma_1 - 523.3\sigma_2 + 785.5\sigma_3 - \\ &\quad 1802.9\xi_1 - 6840.4\xi_2 - 7271.4\xi_3 + \\ &\quad 24.6l - 143.3l^2 + 42.1l^3 \end{aligned}$$

On voit que les coefficient des paramètres ξ_i sont assez importants, mais les corrélations ne s'améliorent que très peu. Généralement on peut dire que les corrélations ne nous donnent pas vraiment des bons indices sur la forme des découpages. Aucun des chiffres n'est donc tellement particulier que l'on puissent le prendre comme critère.

9.3.2 Cas hétérogène

Les exemples ci-dessus utilisent une conductivité uniforme. Quelques tests ont été également faits pour une plaque avec une conductivité inhomogène. Comme premier cas, nous avons pris un $K = 0.001W/m^2K$ dans la moitié gauche et un $K = 1W/m^2K$ dans la moitié droite. Pour un partitionnement binaire (deux sous-domaines), on peut remarquer que le temps de calcul nécessaire augmente pour toutes les configurations de 2 à 10 % par rapport au cas homogène. Néanmoins, il y a une exception remarquable. Si la frontière entre les deux partitions suit exactement la discontinuité au milieu de la plaque, la simulation s'accélère. Ce résultat est une confirmation des résultats théoriques obtenus pour le cas monodimensionnel. Couper un système à l'endroit où se trouve une césure physique rend la simulation plus efficace.

Ce dernier résultat pouvait être confirmé par d'autres tests. Si l'on introduit des discontinuités qui suivent des partitions existantes, le temps de calcul diminue généralement entre 3 et 12 %. L'accélération est plus petite, si le partitionnement est régulier avec des grandes zones. Elle est plus grande pour un partitionnement irrégulier et des petits sous-domaines.

9.4 Tests sur une simulation dynamique

Finalement, nous avons testé l'influence du découpage sur la vitesse de la simulation à l'aide d'un exemple dynamique. Nous considérons une pièce simple carrée comme un problème bidimensionnel (fig. 9.9). Elle est composée de trois murs, quatre coins et le quatrième mur est remplacé par une fenêtre par laquelle entre le rayonnement solaire. L'environnement de la pièce se trouve à température constante, $19°C$, partout.

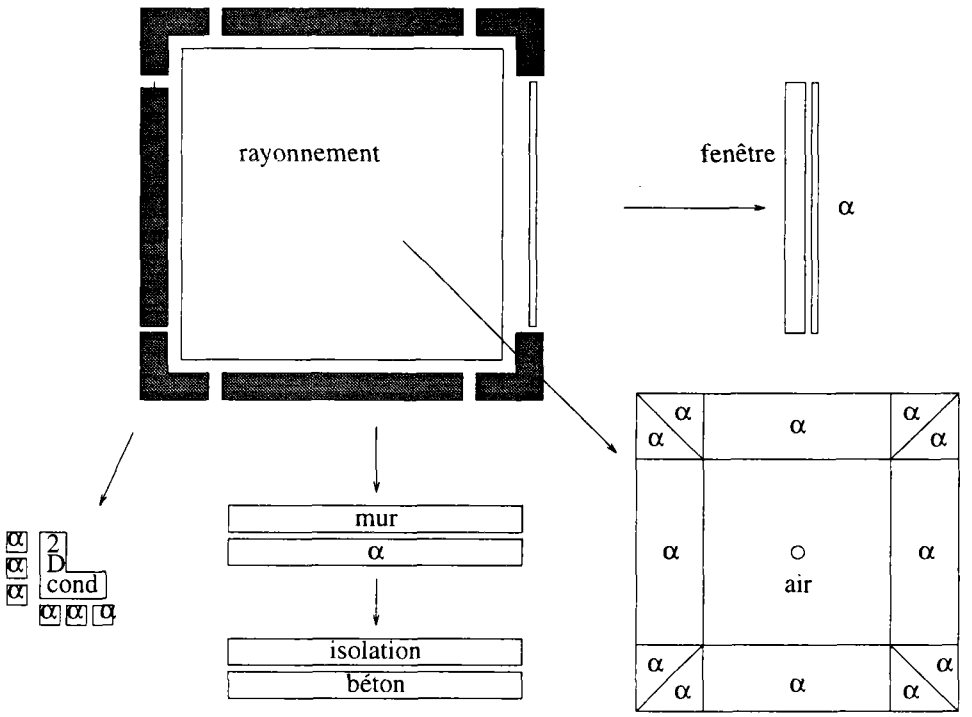


FIG. 9.9 - *La pièce et un découpage possible.*

Au niveau physique, nous ne supposons une conduction bidimensionnelle que dans les coins ; les murs sont suffisamment bien décrits par une conduction monodimensionnelle. La température interne est considérée comme homogène, un seul nœud capacitif le représente. Des coefficients d'échange de chaleur décrivent le flux entre les surfaces des murs et l'air intérieur. À l'exception du rayonnement solaire, le rayonnement entre les surfaces est infrarouge, et la fenêtre se comporte donc pour ce phénomène comme un mur. Le flux solaire est absorbé en partie par la fenêtre, mais la plupart est reçu par la surface à l'opposé de la fenêtre (mur nord). Le rayonnement solaire absolu varie en fonction

des données météorologiques.

Les niveaux mathématiques et algorithmiques ne sont pas détaillés pour les modules élémentaires de cet exemple. Ici notre préoccupation principale est l'influence du découpage sur le temps de calcul nécessaire. Néanmoins, on peut remarquer un point important de la modularité de Motor-2, car les modèles utilisés ont des origines très différentes. Nous utilisons les modèles déjà utilisés de coefficient d'échange et de conduction monodimensionnelle, mais aussi un modèle modal [52] pour la conduction bidimensionnelle dans les coins, et un modèle de rayonnement [17].

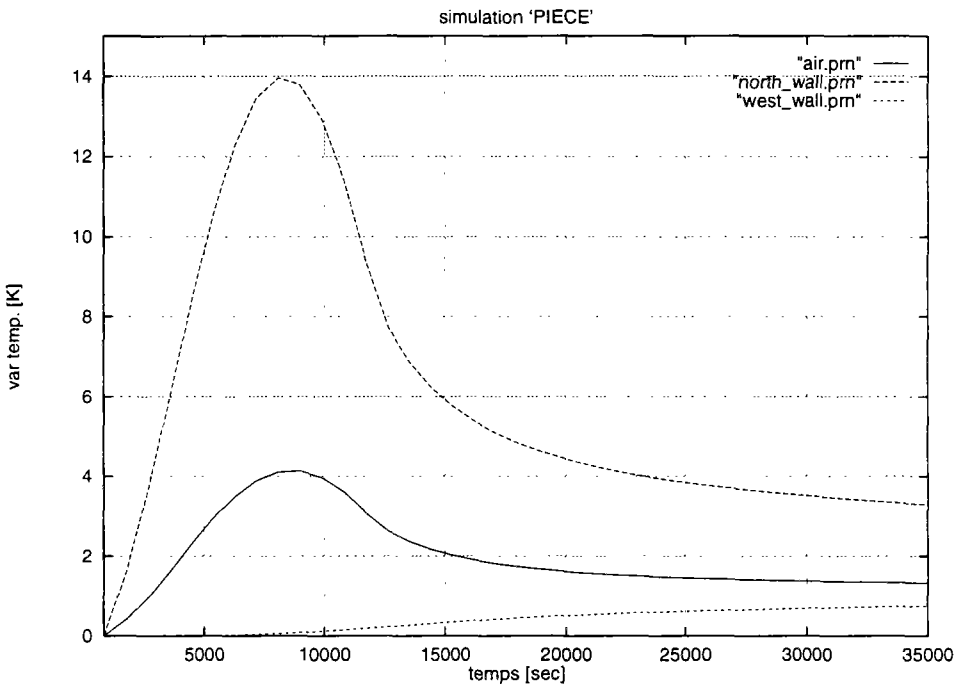


FIG. 9.10 - *Résultats de quelques données de la simulation.*

Nous avons mesuré le temps nécessaire pour une simulation de 46.800 secondes avec un pas de temps de 900 secondes. Le flux absorbé dans la fenêtre et sur la face nord du local varie en sinusoïde. C'est le moteur de la dynamique de notre système. Dans la figure 9.10 les évolutions de quelques variables sont tracées. On remarque bien le chauffage rapide de l'air intérieur qui n'a qu'une très petite capacité. La ligne continue montre l'évolution de la température de la surface nord. Ici est absorbé le flux solaire qui chauffe l'air et en moindre degré le béton du mur. À l'intérieur du mur ouest entre la couche de béton et celle d'isolant, l'augmentation de la température ne commence de se faire sentir qu'après 10.000 secondes (la ligne pointillée).

La figure 9.11 montre deux découpages différents. L'arbre PROF à gauche est volontairement ramifié profondement. Nous avons des modules terminaux jusqu'au niveau trois au-dessous du module principal. À l'opposé de ce découpage se trouve la configuration PLAT à droite, où tous les modules terminaux sont directement couplés par le module principal. Nous avons donc une structure plate

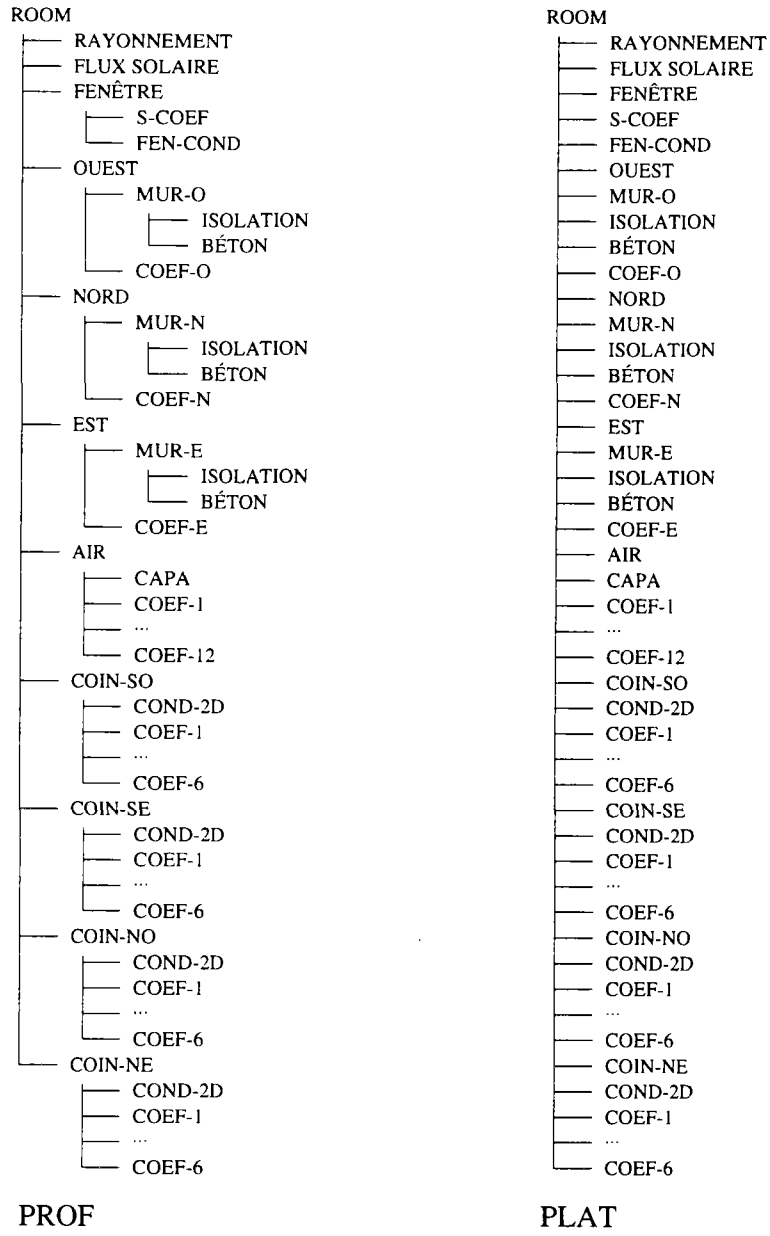


FIG. 9.11 - *Différents découpages extrêmes de la pièce.*

comme on la trouve généralement dans les environnements non-hiérarchisés.

Trois autres configurations ont été testés également. Leur découpage est montré dans la figure 9.12. Le degré de ramification des trois configurations se trouve entre les deux extrêmes précédents. Dans la configuration AIR-FEN-PLAT à gauche, nous avons gardé la structure hiérarchique. Seuls les sous-modules de la fenêtre et de l'air interne ont été ramenés au plus haut niveau. La configuration COIN-PROF au milieu est comparable à la structure plate, mais les sous-modules des coins ont été laissés hiérarchisés. Dans la dernière configuration COIN-MUR-PROF à droite, les sous-modules du mur nord ont été résolus et mis au plus haut niveau, pour le reste, c'est la même configuration que

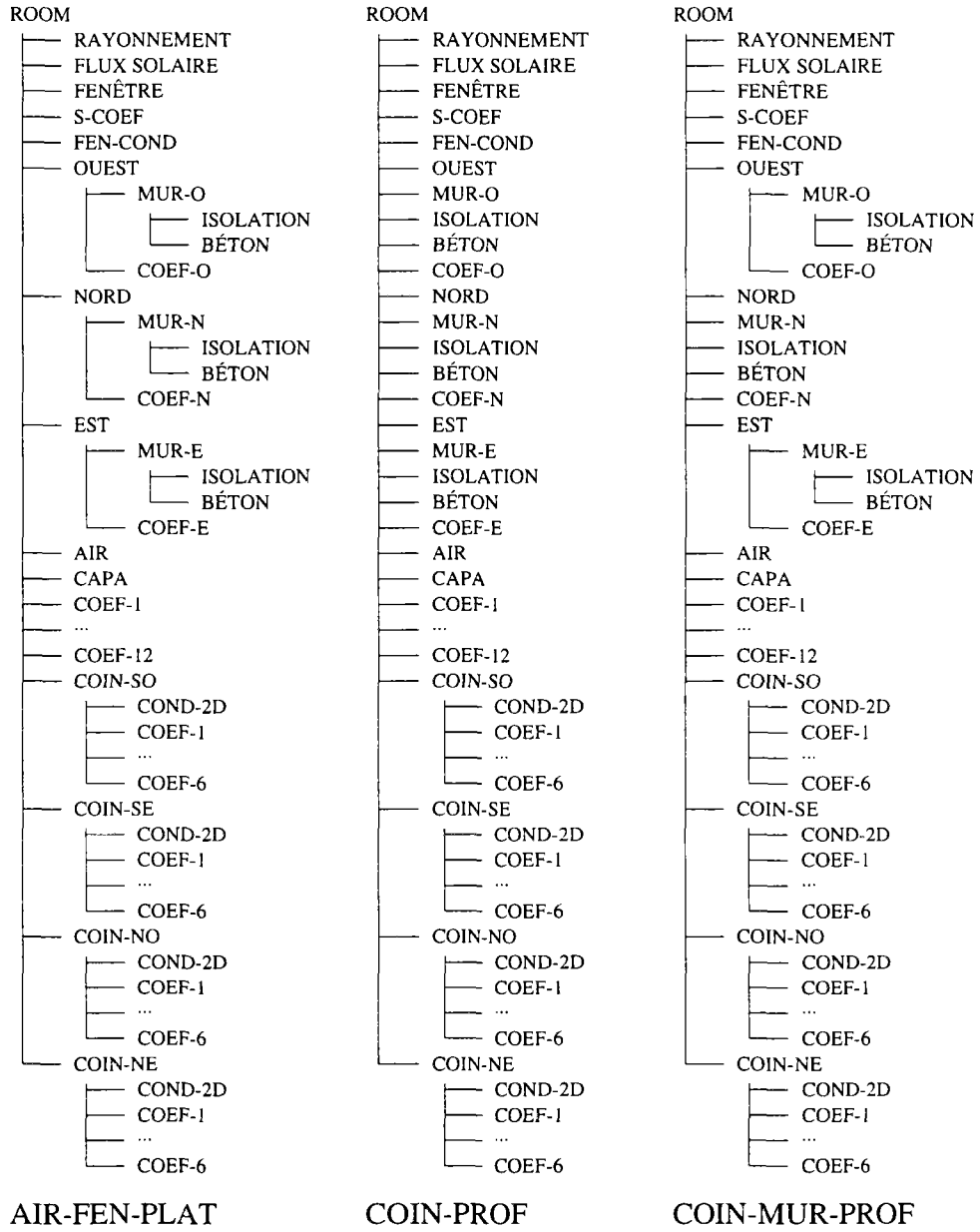


FIG. 9.12 - *Découpages intermédiaires de la pièce.*

AIR-FEN-PLAT.

	PLAT	PROF	AIR-FEN-PLAT	COIN-PROF	COIN-MUR-PROF
temps [sec]	198,4	166,1	152,1	163,9	153,6
gain [%]		16,1	23,3	17,4	22,6

FIG. 9.13 - *Les temps utilisateur des simulations pour les différentes configurations.*

Pour toutes les configurations, nous avons appliqué la méthode GLOBALE pour la résolution des modules composés. Les temps d'exécution nécessaires

pour chacune des configurations sont listés dans le tableau 9.13. On remarque tout de suite que la configuration où tous les modules sont mis à plat, est la plus lente. C'est certainement dû à l'évaluation numérique d'une grande matrice *Jacobienne*, bien que la plupart des interfaces ne varient pas. La configuration PROF avec beaucoup de ramifications est déjà 16 % plus rapide. Mais toutes les configurations intermédiaires sont encore plus rapides. Le gain de vitesse peut atteindre 23,3 % pour la configuration AIR-FEN-PLAT. À peu près le même résultat est obtenu avec la configuration COIN-MUR-PROF. Dans les deux cas, les sous-modules des coins et les murs des cotés n'ont pas été mis à plat. Les valeurs numériques ne changent que très peu dans ces sous-modules. La matrice glabale de dépendances n'est donc pas beaucoup perturbée par l'omission de ces modules.

Pour bien accélerer un tel système, il faut savoir *a priori* où se passe les calculs et quels sont les endroits qui restent plus ou moins stables. Si l'on arrive à bien séparer ces régions, on peut espérer des gains de vitesses considérables.

9.5 Conclusion

Les expérimentations effectuées sur la simple plaque aussi bien que sur l'exemple du local montrent parfois une bonne correspondance entre un découpage guidé par des considérations techniques et un découpage en vue d'une efficacité numérique. Néanmoins, quelques configurations singulières parmi les découpages de la plaque nécessitent des coûts importants sans que nous sachions d'où vient ce temps de calculs excessif. Les corrélations entre les vrais coûts et les fonctions construites ne sont pas suffisamment grandes pour conclure avec certitude un schéma de découpage efficace.

Il faut noter que d'autres approches modulaires de la simulation ne s'appuient pas du tout sur des considérations techniques pour la mise en œuvre de la modularité. L'environnement Spark contient son propre algorithme interne pour la réduction des équation. Ceci peut aussi être vu comme comme une sorte de « découpage ».

Sous le nom de « décomposition de domaine» (*domain decomposition*) s'est établie récemment une brache de mathématiques et génie d'ingénieur qui étudie une classe de méthodes numérique pour obtenir des solutions par combinaison des solutions des sous-domaines. On s'y intérroge également sur les sous-structurations adéquates pour des grands systèmes d'équations différentielles partielles. Ces équations ont souvent la même structure qu'elles proviennent d'une modélisation par différences ou par éléments finis. La combinaison des solutions des composants d'une physique différente comme c'est le cas dans Motor-2 n'est pas traité.

Le problème du découpage est d'un intérêt évident, notammment dans la conception des programmes liés à une architecture d'ordinateurs à haute performance.

Chapitre 10

Conclusion et perspectives

État actuel et connaissances

Les systèmes thermiques développés aujourd'hui sont de plus en plus grands et complexes. Aussi, la difficulté de comprendre et de maîtriser cette complexité augmente. Très souvent des études détaillées sont nécessaires pour exactement déterminer les besoin de chauffage d'un bâtiment, d'un processus industriel, etc. Comme ces études sont nécessaires dans la plupart des cas avant de construire le bâtiment ou l'usine, on a recours à la simulation du système par le biais d'un modèle. De nombreuses conférences internationales et de nouvelles revues traitant de ce thème montrent l'importance croissante de la modélisation et de la simulation.

Les approches et concepts pour le développement d'un environnement de simulation sont introduits dans cette thèse. Dans la conception de notre environnement, nous considérons les étapes de l'*approche systémique* et surtout les derniers développements au niveau du génie logiciel comme la *Conception Orientée Objet* et l'application du parallélisme qui permet de profiter des réseaux d'ordinateurs et d'une évolution accélérée des ordinateurs parallèles.

Un des objectifs de cet environnement est la capitalisation des efforts de recherche dans le domaine de la modélisation pour alléger la tâche de l'utilisateur. Il existe d'une part des chercheurs qui développent des modèles sophistiqués, d'autre part des utilisateurs qui veulent obtenir des résultats sans trop entrer dans les détails de l'implémentation. Nous avons donc développé un environnement de simulation qui tâche de fonctionner comme une jonction entre les deux pôles.

Résumé du travail présenté

Dans le travail présenté, une description à plusieurs niveaux d'abstraction a été développée. Le systèmes et ses composants passent d'une couche technique

aux niveaux physique et mathématique pour ensuite arriver au niveau algorithmique, tout en gardant une représentation informatique le plus près possible d'une image naturelle de l'objet réel.

Le concept de base qui permet la séparation des rôles d'un utilisateur de celui d'un développeur est l'informations d'accès limité. Seule une couche de spécifications est visible à l'utilisateur ; la réalisation lui est cachée. C'est également un des points principaux de la *Conception Orientée Objet.* De cette façon les dépendances et la communication entre les modules sont bien définies. Une modification dans l'implémentation n'entraîne pas de changement hors du module.

Pour effectuer une étude de système avec notre environnement de simulation, l'utilisateur doit d'abord découper son système en plusieurs étapes. Niveau par niveau, il crée un graphe de couplage entre les composants du système. Ensuite il fournit une description de cette analyse par un ensemble de fichiers contenant les paramètres des modules terminaux et des connexions hiérarchiques entre les composants. Le logiciel de simulation attribue à chaque composant son propre code de calcul indépendant, potentiellement exécutable simultanément avec les autres composants. C'est notre approche d'utiliser le parallélisme par extension naturelle des composants vers des composants actifs.

L'essentiel de la simulation passe par une résolution des conditions de raccordement entre les modules. À chaque niveau de la hiérachie, nous créons des modules composés. À l'intérieur de ces modules les *interfaces* rassemblent les équation de raccordement entre les modules. Tous les sous-modules connectés par des interfaces sont déclenchés d'une manière parallèle. Les calculs des sous-modules sont alors effectués simultanément. Nous avons défini des points d'entrée qui permettent la communication entre les interfaces et les modules. Différentes méthodes algorithmiques peuvent résoudre les relations des interfaces.

Notre environnement de simulation n'est pas uniquement un logiciel pour la résolution des systèmes d'équations différentielles. Les modèles développés par des chercheurs sont les plus généraux possibles. Le langage habituel pour les exprimer sont des relations mathématiques, mais nous ne sommes pas limités à cette représentation. Toute expression qui peut être représentée dans un langage informatique est acceptée. Les modèles sont stockés dans des bibliothèques ou plutôt des « *modélothèques* ». Pour transformer le modèle général de base vers un code de calcul, nous avons établi des représentations intermédiaires qui arrivent finalement au niveau d'inclusion dans le logiciel de résolution.

À l'aide de trois exemples, les possibilités de l'environnment de simulation sont illustrées. Le découpage hiérachique et sa conservation au niveau de la résolution suscite l'inérrogation sur l'efficacité numérique pour des connexions d'un point de vue technique.

Réalisations et perspectives

L'objectif d'un environnement modulaire de simulation est atteint. Son approche de parallélisation qui affecte à chaque composant sa propre file d'exécution est une extension naturelle de la *Conception Orientée Objet.* Elle permet l'usage aussi bien des ordinateurs parallèles que des ordinateurs inter-connectés dans un réseau. La faisabilité d'une telle approche est maintenant évidente. L'application du simulateur dans un laboratoire de recherche a montré à la fois son utilité et sa fiabilité.

Il reste cependant quelques points qui méritent d'être approfondis. Parmi ces points, il y a en première ligne la représentation informatique des modèles aux différents niveaux. GRAMSYM est certainement un premier pas prometteur, mais il ne couvre pas encore toutes les bibliothèques du projet SYMBOL. L'usage d'une représentation basée sur une norme qui s'est déjà établie ailleurs comme le format NMF par exemple facilite l'intégration des modèles provenant d'autres environnements et l'échange de modèles entre les utilisateurs et chercheurs. Par ailleurs, l'environnement doit permettre l'importation des informations provenant d'un plan d'architecte comme la géométrie ou la composition des matériaux. Un changement du format de représentation doit également revoir la structure des fichiers d'entrée. Leur usage paraît parfois un peu lourd, bien que le format soit bien structuré et très lisible.

Un deuxième point d'amélioration possible est l'exploitation du parallélisme. Une charge de travail égale des processeurs physiques est souhaitable. Dans un premier temps, l'environnement de simulation doit permettre à l'utilisateur de spécifier le nombre d'ordinateurs à utiliser et leurs puissances relatives pour mieux exploiter les ressources de calcul disponible. Un deuxième pas pourrait automatiser cette distribution en fonction des puissances et des charges actuelles des ordinateurs.

L'interface utilisateur est également à rendre plus conviviale. Ce point ne concerne pas seulement Motor-2, mais aussi d'autres environnements de simulation. Une saisie du système et une modification des paramètres d'une façon intéractive dans un environnement graphique faciliterait l'usage de la simulation. On peut connecter et découper les composants par une simple action avec le matériel de saisie (souris, stylo de lumière, etc). Dans l'avenir, cette interface graphique peut être équipée d'une intelligence artificielle d'une part pour guider le découpage, et d'autre part pour une génération automatisée de modèles. Cette génération peut être ontenue d'un groupement de modèles plus élémentaires avec prise en compte de modifications nécessaires pour la connexion. On peut également imaginer des modèles auto-adaptables qui se modifient en fonction de la précision choisie ou des efforts de calcul. L'emploi de ces méthodes automatiques demande une plus grande prudence à l'utilisateur pour bien vérifier les modèles et le raisonnement de leur génération.

Annexe A

Nomenclature

symbol	unité	description
A, B, C		noms de modules
$\boldsymbol{C}$		matrice globale de raccordement
c	$[\frac{J}{kg}]$	capacité calorifique
$\boldsymbol{E}^A$		vecteur de toutes les entrées du module A
$\boldsymbol{e}_b^A$		vecteur des entrées du module A qui sont liées à l'interface b
h	$[\frac{W}{K}]$	conductance
$\boldsymbol{J}$		matrice des dérivées, dite la *Jacobienne*
P	$[W]$	puissance
R^A		fonction d'une frontière qui calcule les sorties locales s en fonction de toutes les entrées $\boldsymbol{E}$
$\hat{R}^A$		fonction d'un module qui calcule toutes les sorties S en fonction de toutes les entrées $\boldsymbol{E}$
$\tilde{R}^A$		fonction d'une frontière qui calcule les sorties locales s en fonction des entrées locales $\boldsymbol{e}$
$\boldsymbol{S}^A$		vecteur de toutes les sorties du module A
$\boldsymbol{s}_b^A$		vecteur des sorties du module A qui sont liées à l'interface b
T	$[K]$	température
t	$[s]$	temps
t_i	$[s]$	instant i
$\boldsymbol{\eta}^A$		état interne du module A
Φ_b^A		frontière du module A connectée à l'interface a
ϕ	$[W]$	flux de chaleur
λ	$[\frac{W}{mK}]$	conductivité
ρ	$[\frac{kg}{m^3}]$	masse volumique

Annexe B

Mode d'emploi de **Motor-2**

B.1 Manuel de l'utilisateur

Résumé

Ce document décrit l'utilisation du logiciel Motor-2 qui est un environnement modulaire de simulation de systèmes thermiques. Après une précision terminologique, les étapes conduisant à la simulation seront décrites.

B.1.1 Présentation

Motor-2 est un logiciel de simulation à structure modulaire. Il est surtout conçu pour des simulations de modules dont le comportement dépend continuement du temps.

Un des grands avantages du projet SYMBOL est la séparation entre l'utilisation des modèles et leur développement. L'utilisateur peut décrire son système par un raccordement d'éléments provenant d'une bibliothèque de modèle. Cette distinction entre développeur et utilisateur est également rapportée dans ce manuel qui se structure en deux parties. La première partie décrit comment doit un utilisateur préparer une simulation. Une deuxième partie B.2 explique les démarches nécessaires à l'extension de la bibliothèque de Motor-2.

Dans cette première partie, nous expliquons d'abord le processus de description d'un problème sur ordinateur et le lancement des simulations de ce problème avec Motor-2. Il suit une description détaillée des structures des différents fichiers d'entrée, des fichiers de configuration et du fichier 'log'. Mais tout d'abord, nous allons spécifier la signification de quelques mots qui seront souvent utilisés.

B.1.2 Terminologie

Un *système* est défini comme un ensemble de *modules* qui sont *raccordés* entre eux de manière à ce que l'ensemble décrive un certain problème. Pour

étudier par exemple la conduction monodimensionnelle à travers un mur, on peut raccorder une couche d'isolant avec une couche de béton qui sont sollicitées avec la température intérieure et extérieur par des résistances thermiques. Ce *système* est alors composé de six *modules*: un pour chacune des températures imposées (variables), un pour chacune des résistances et un pour chacune des couches. Cette subdivision en plusieurs modules renforce le caractère modulaire de Motor-2. La simulation du système entier passe par la simulation des modules raccordés et la résolution des raccordements.

La simulation d'un module seul passe par des *entrées* et des *sorties* du module. L'utilisateur doit alors spécifier les *pattes* (« pins ») de connexion avec ses paramètres (nombre, noms, type, valeur initiale, ...). À un niveau supérieure les *pattes* d'un module sont rassemblés dans des *frontières*. Une *frontière* est une collection de *pattes* qui vont logiquement (physiquement, topologiquement) toujours ensemble. Cela peut être une température et un flux de chaleur ou un débit massique avec son enthalpie.

Le comportement d'un module dépend de son type et des valeurs de ces paramètres. Le type – ou mieux – la *classe du module* fixe le nombre, l'ordre et la signification des paramètres des ses instances, des modules. L'exemple ci-dessus a deux modules COEF_INT et COEF_EXT qui appartiennent à la classe COEF. Un *module* de cette classe attend un seul paramètre qui est le coefficient d'échanges en W/K.

Un *module* peut aussi être composé de plusieurs *sous-modules*. Dans notre exemple, on pourrait composer un module MUR des sous-modules BÉTON et ISO. Dans ce cas la *classe de module* s'appelle LINKUP et l'utilisateur doit spécifier dans la partie de paramètres du module les noms des sous-modules (`Béton`, ISO), le raccordement entre eux par des *frontières* (contact parfait entre la face extérieure du BÉTON et la face intérieure de l'ISOlation) et la visibilité des frontières de sous-modules pour les autre modules du système (la face intérieure du MUR et la face intérieure du BÉTON).

B.1.3 Comment faire une simulation?

Préparations pour une simulation

Après une analyse de son problème, l'utilisateur identifie les différents composants de son système et les relations entre ces composants. Chacun des composants est représenté par un module pour lequel il faut un fichier qui décrit le module en termes d'*entrées/sorties* et *frontières*. La dernière partie de ce fichier contient un mot-clef qui spécifie la classe à laquelle le module appartient (le modèle) et des paramètres de cette classe. Chaque module composé est également décrit par un tel fichier. Le système entier est décrit par un module composé principal. Ses variables d'entrée sont considérées comme des constantes, les sorties sont negligées. C'est le nom de cette unité principale qui est à passer à Motor-2 au moment du lancement. Ce module fait référence à ses sous-modules qui à leur tour incluent ses propres sous-modules et de cette façon toute la structure du problème est lue récursivement par Motor-2.

Lancement de Motor-2

Outre des fichiers de module, Motor-2 nécessite les fichiers de configuration `sim.time` pour spécifier le début et la fin de la simulation et le pas de temps initial, et `link.cfg` où on indique quelques paramètres de l'algorithme numérique. Dans la version actuelle une configuration pour le « *debugging*» est possible avec le fichier `debug.cfg` qui est lu s'il existe.

Une fois tous les fichiers de modules et de configuration préparés, on peut lancer le simulateur Motor-2. Il faut passer le *nom* du module principale de son problème comme argument. Supposons que le fichier `wall.smd` contienne la description de notre exemple, on tape alors

```
% motor2 wall
```

et le programme commence à lire les fichiers de configuration, des variables d'environnement (voir plus loin) et finalement le fichier `wall.smd`. Ce fichier contient toutes les information pour pouvoir lire tous les sous-modules avec leurs paramètres.

```
Usage: motor2 [-vldhp] main-object,
 -v[0-4]       : verbosity.
    0          :     quiet, no messages at all (beside errors).
    1          :     normal (default).
    2          :     some 'useful' messages.
    3          :     some more messages.
    4          :     tells you everything.
 -l[0-2][file] : log. Write status to a log file (symbol.log).
    0          :     don't create a log file.
    1          :     write status at every time step (default).
    2          :     write status at every iteration step.
 -ppath        : path. Look for modules in directory 'path'.
                      (overwrites value of MOTOR2_PATH.)
 -d            : display. Show tree with dependencies in another window.
 -tfile        : tasking.
 -h            : help. Show this message.
```

FIG. B.1 - *Le message de Motor-2 en cas d'un appel sans argument.*

Si on tape simplement `motor2` sans argument on voit afficher une courte description des options et le programme s'arrête (fig. B.1). Il existe les options `-v`, `-l`, `-p`, et `-d`. Elles ont les significations suivantes:

`-v#` (verbosity level) Cette option règle le nombre de message que le programme Motor-2 écrit à l'écran (dans `stderr`). Le chiffre `0` supprime totalement les messages. Seuls les messages d'erreur passent encore. Le niveau `1` pose des questions quand c'est nécessaire, le niveau `2` affiche des message sur la progression de la simulation, au niveau `3` il met un message après chaque fichier lu et chaque pas de temps, et le niveau `4` ajoute encore des messages pour le « *debugging*».

`-l#name` (logfile, journal de la simulation). Cette option contrôle le journal de la simulation. Un chiffre comme premier argument spécifie la précision du journal et un argument « string » indique le nom de fichier.

La valeur `0` supprime la création du fichier journal. Motor-2 prend `1` comme valeur par défaut et il écrit la date, l'heure et sa sa configuration momentanée au début du fichier. Ensuite le programme demande après chaque pas de temps à tous les modules qu'ils écrivent leur état dans ce fichier. À la fin de la simulation le fichier est conclu avec l'heure actuelle. Avec la valeur 2 les modules écrivent leur état à chaque pas d'itération dans le journal.

Un nom derrière le chiffre indique le nom du fichier journal. En cas d'omission Motor-2 écrit dans `symbol.log` par défaut.

`-ppath` (path) On indique ici le répertoire où Motor-2 cherche les fichiers des modules. Tous les autres fichiers (de configuration, de météo, le journal) ne sont pas concernés par cette option. L'usage de la variable d'environnement `MOTOR2_PATH` qui sert à la même chose est recommandé. Mais elle n'est pas prise en compte si l'option `-p` est utilisée.

`-tfile` (tasking) Avec cette option, Motor-2 écrit à chaque appel d'une tâche la valeur de l'horloge interne dans le fichier spécifié. Ce fichier peut ensuite être lu par l'utilitaire Upshot pour l'affichage de l'ordre et de la durée des activations des tâches.

Toute autre lettre est considérée comme une erreur, elle fait afficher le mode d'emploi et Motor-2 termine sans action.

Les fichiers nécessaires

Il y a trois type de fichiers que Motor-2 lit. Le premier groupe comprend les fichiers qui sont propres à la simulation. Les fichiers `*.smd` décrivent le système à simuler et le fichier `sim.time` contient le début, la fin et le pas de temps de la simulation. Un journal de la simulation est écrit dans le fichier `symbol.log`.

Les fichier de configuration spécifient quelques options dynamiques de Motor-2. Avec le fichier `link.cfg`, on a un moyen de contrôler l'algorithme numérique comme l'erreur relative et l'erreur absolue. Le fichier `debug.cfg` permet l'affichage des messages de bon fonctionnement de Motor-2. Il n'est pas nécessaire.

Les fichiers `pins.list`, `frontiers.list`, `contents.list` et `interfaces.list` font partie de l'installation de Motor-2. Ils définissent les types connus pour des *pattes*, des *frontières*, les *modèles* et les *interfaces*. Par defaut, Motor-2 cherche ces fichiers dans le répertoire `/usr/local/lib/symbol`. La variable d'environnement `M2_INST_PATH` peut indiquer un autre endroit.

fichiers de simulation	fichiers de configuration	fichiers d'installation
`*.smd`	`link.cfg`	`pins.list`
`sim.time`	`debug.cfg`	`frontiers.list`
`symbol.log`		`contents.list`
		`interfaces.list`

FIG. B.2 - *Les fichiers de Motor-2*

B.1.4 Structure des fichiers

Motor-2 lit les fichiers de l'installation, de la configuration et des fichiers qui sont propres à la simulation. Pendant la simulation, il écrit en général un fichier journal et probablement des fichiers résultats. Tous les fichiers utilisés par Motor-2 suivent quelques règles communes que nous présentons ici.

Au niveau lexical tous les caractères du jeu ASCII sont permis. Le programme ne distingue pas les minuscules des majuscules. Tout nom interne est représenté par des caractères minuscules. Néanmoins, quelques caractères ont des significations spéciales. C'est le supérieur (>), le trait vertical (|), le double tiret (--), et le saut de ligne. Il y a une différence si une information est écrite sur deux lignes ou dans une seule.

Les commentaires sont permis dans les fichiers SYMBOL, ils sont même souvent utilisés dans les fichiers générés (par des pré-processeurs ou par Motor-2 même). Un commentaire commence par un double tiret (--) et se termine à la fin de la ligne. Un commentaire qui se trouve au début de tous les fichiers est le suivant[1]:

```
--                                -*- Symbol -*-
-------------------------------------------------------------------------------
```

Les commentaires sont une aide pour l'utilisateur qui veut lire (et écrire) des fichiers. Ces informations ne sont pas pris en compte par la machine.

Les informations pour Motor-2 dans les fichiers sont structurées en blocs qui sont eux-mêmes structurés en champs. Un champ est le texte inclus entre deux traits verticaux comme par exemple dans cette ligne qui définit une *patte* et qui contient quatre champs:

```
| Temp_Outside        | Temperature         | 0.00000E+00  |        K          |
```

Les « whitespaces » (blancs et tab) au début et à la fin d'un champ ne sont pas pris en compte. Le texte qui reste est limité à 20 caractères.

1. La partie de commentaire `-*- Symbol -*-` est l'indicatif pour l'éditeur **emacs** qui se met alors en mode **symbol** pour éditer ce fichier

Plusieurs lignes de ce type forment un bloc. Pour en indiquer la fin, il faut une ligne qui commence avec le caractère supérieur (>). Le reste de cette ligne est sauté. Voici le bloc des *pattes* d'entrée pour le module ISO

```
--                              <<  Input  >>
--       Name        |        Type         |   Vdefault    |       Unit        |
-------------------------------------------------------------------------------
| Temp_Inside        | Temperature         | 0.00000E+00    |         K         |
| Temp_Outside       | Temperature         | 0.00000E+00    |         K         |
>------------------------------------------------------------------------------
```

Les trois premières lignes sont des commentaires. Les lignes 4 et 5 contiennent les déclaration de deux *pattes* `Temp_Inside` et `Temp_Outside`. La dernière ligne indique la fin du bloc *liste de pattes d'entrée.*

Structure des fichiers de modules

Le nom d'un fichier de module au niveau du système d'exploitation se déduit du nom du module où tous les caractères sont écrits en minuscules. Au nom est ajoutée l'extension `.smd`. Motor-2 cherche tous les modules dans le même répertoire qui est indiqué par l'option `path` (voir en haut) ou par la variable d'environnment `MOTOR2_PATH`.

Le contenu d'un fichier a quatre parties. D'abord, il y a la partie administrative qui contient trois lignes (voir fig. B.3)

1° un texte indiquant la version de la structure SYMBOL utilisée,

2° un texte qui dit d'où vient ce fichier (p.e. `Hand` veut dire qu'il a été écrit à la main, sinon c'est le nom du pré-processeur),

3° et finalement le nom du module.

Plusieurs champs peuvent encore suivre jusqu'à la marque de la fin du bloc, mais ne sont pas pris en compte. Motor-2 avertit l'utilisateur si le numéro de la version ne correspond pas avec sa propre liste des versions lisibles (4.1 et 4.2) ou si le nom ne correspond pas au nom du fichier.

```
--                              -*- Symbol -*-
-------------------------------------------------------------------------------
| 4.2                | Version
| Hand               | Origin
| W_Insul            | Object Name
>------------------------------------------------------------------------------
```

FIG. B.3 - *L'entête d'un fichier SYMBOL.*

La deuxième partie du fichier définit les pattes de communication, les *pins* (voir fig. B.4). Il y en a trois types principaux pour les différentes directions : des entrées (`Input`), des sorties (`Output`) et des pattes d'observation (`Observ`). Pour les lire Motor-2 attend alors trois blocs consécutifs pour ces trois directions. Les

variables de la direction *Observ* sont reconnues, mais elles ne sont pas utilisées par Motor-2.

Une ligne d'un tel bloc constitue la définition d'un *patte* pour le module. Dans une ligne, les premiers quatre champs ont les significations suivantes : nom de la patte, type physique de la patte (par exemple `Mass_Flow` (débit massique)), sa valeur initiale, et son unité (kg/s). Le reste de la ligne est ignorée. Seuls les types de pattes physiques sont permis dont le nom est mentionné dans le fichier d'installation `pins.list`.

```
--                              << Input  >>
--       Name        |        Type        |   Vdefault   |       Unit        |
-------------------------------------------------------------------------------
| Temp_Inside       | Temperature        | 19.0000E+00  |                   |
| Temp_Outside      | Temperature        | 19.0000E+00  |                   |
>------------------------------------------------------------------------------
--                              << Output  >>
--       Name        |        Type        |   Vdefault   |       Unit        |
-------------------------------------------------------------------------------
| Flux_Inside       | Heat_Flux          |  0.0000E+00  |                   |
| Flux_Outside      | Heat_Flux          |  0.0000E+00  |                   |
>------------------------------------------------------------------------------
--                              << Observ  >>
--       Name        |        Type        |   Vdefault   |       Unit        |
-------------------------------------------------------------------------------
>------------------------------------------------------------------------------
```

FIG. B.4 - *Definition de deux* pattes *d'entrée et deux* pattes *de sortie.*

Suit le bloc des *frontières*. Nous avons défini plus haut une *frontière* comme un ensemble de *pattes* associé avec un type. Le premier champ est le nom de la *frontière*, le deuxième contient le type de la *frontière*. Au troisième champ succède la liste des pattes qui constituent cette *frontière*. Les *pattes* se trouvent toujours dans le troisième champ d'une ligne. Le premier champ de la ligne suivante contient

- ou rien (dans ce cas le troisième champ contient la patte suivante de la *frontière* actuelle),
- ou le nom de la *frontière* suivante, la liste de la frontière actuelle est terminée,
- ou le caractère '>' qui indique la fin de la liste des *frontières*.

Cela paraît compliqué, mais c'est très lisible pour un être humain. Ici comme pour les *pattes*, les types des *frontières* doivent être déclarés pour Motor-2 dans un fichier d'installation `frontiers.list`.

La dernière partie contient un mot-clef pour indiquer la *classe de module* (le modèle) à laquelle ce *module* appartient (voir fig. B.6). Ce mot-clef n'est pas nécessairement dans un champ (entouré par des traits verticaux) pour des

```
--                              << Frontiers >>
--      Name          |        Type         |    Pins           |
-------------------------------------------------------------------------------
|Inside               |First_Kind           |Temp_Inside        |
|                     |                     |Flux_Inside        |
|Outside              |First_Kind           |Temp_Outside       |
|                     |                     |Flux_Outside       |
>------------------------------------------------------------------------------
```

FIG. B.5 - *La déclaration de deux frontières de type* `First_Kind`. *La frontière* `Inside` *est l'ensembles des pattes* `Temp_Inside` *et* `Flux_Inside`.

raisons de compatibilité avec des versions précédentes. La structure des paramètres qui suivent le mot-clef dépend obligatoirement de la *classe de module.* Cela peut être une seule valeur réelle comme pour la *classe* des résistances thermiques, mais cela peut être aussi bien des grandes matrices pour un *module* de la *classe* MODALe. Dans la deuxième partie B.2, nous expliquons comment un nouveau modèle spécifie la forme de ces paramétres.

```
| Coef        |
---------------
|    0.04     |
>--------------
```

FIG. B.6 - *La dernière partie du fichier contient un mot clé (ici* `Coef`*) pour indiquer la* classe du module *et les paramètres (ici une valeur réelle).*

Structure des fichiers de modules composés

Les trois premiers blocs, la partie pour la gestion, les *pattes*, et les *frontières*, sont les mêmes pour tous les modules qu'ils soient terminaux ou composés. La dernière partie doit être définie différemment pour chaque classe. Comme les modules composés nous sont indispensables, ceci a été fait pour cette classe.

Deux blocs sont nécessaires pour la description des connexions internes et des connexions avec les frontières extérieures.

Le premier bloc contient une liste des *interfaces* pour les relations entre les sous-modules (fig. B.7). Une *interface* est constituée d'un *type* et d'une liste de *links* où un *link* est l'ensemble d'un *module* et d'une de ses *frontières.* La liste des *links* contient donc toutes les *frontières* des sous-modules qui sont à connecter. Dans le fichier cela se présente de façon suivante :

- le premier champ d'une ligne indique le *type* de l'*interface.*
- le deuxième champ contient le nom du sous-module et
- le troisième champ sa frontière.

La liste des *links* continue si le premier champ de la ligne suivante est vide. Dans ce cas, les 2e et 3e champs contiennent le prochain module et sa frontière de l'*interface* actuelle. Les deux types d'interface les plus courants sont PERFECT_CONTACT et DIRECT. Une interface de type PERFECT_CONTACT cherche à trouver la bonne valeur des variables d'entrée qui annule la somme des variables de sorties des tous les *links*. Une interface DIRECT distribue la valeur du premier *link* aux autres *links* de l'interface.

```
-------------------------------------------------------------------------
--     Type          |    Modul Name      |    Frontier        |
-------------------------------------------------------------------------
|Perfect_Contact     | Iso                | Inside             |
|                    | Beton              | Outside            |
>-------------------------------------------------------------------------
```

FIG. B.7 - *Le bloc des connexions internes. Le module* MUR *raccorde la frontière* **Inside** *du sous-modules* ISO *avec la frontière* **Outside** *du* BÉTON

Le deuxième bloc d'un module composé contient les extractions des sous-modules. À chaque frontière du module-composé est associée une frontière des sous-modules. Un contrôle est effectué si toutes les frontières des sous-modules sont connectées soit entre elles, soit par équivalence avec une frontière du module « père ».

```
-------------------------------------------------------------------------
-- Father's Frontier | Son's Name         | Son's Frontier     |
-------------------------------------------------------------------------
| Inside             | Beton              | Inside             |
| Outside            | Iso                | Outside            |
>-------------------------------------------------------------------------
```

FIG. B.8 - *Le bloc extract décrit l'équivalence entre des frontières du module composé et des frontières des sous-modules.*

Fichiers de configurations

Pour l'instant, il y un fichier de configuration de Motor-2 qui s'appelle `link.cfg`. Au lancement de Motor-2, ce fichier est cherché d'abord dans le répertoire *config_dir* comme spécifié par la variable d'environnement. Si le fichier n'est pas trouvé dans ce répertoire, il est cherché dans le répertoire courant. Si cette recherche échoue également, Motor-2 prend ses valeurs par défaut. Elles sont montrées dans la figure B.9.

Le fichier `link.cfg` pilote surtout les algorithmes numériques de résolution de raccordements. Comme tous les autres fichiers, il est composés de *champ*, Mais seul le premier champ d'une ligne est pris en compte. Chaque paramètre se trouve sur une nouvelle ligne.

Le premier champ (`Max_Passes`) contient le nombre maximal d'itérations à un niveau. Le deuxième champ (`Jac_Eval`) indique après combien d'itération la matrice des dérivées est à recalculer. Le facteur de retardement (`Rel_Newton`) se trouve au troisieme champ. Il suit le champ (`X-Error`) qui borne l'erreur maximale sur les variables d'itération dans les interfaces. Ensuite le champ (`Y-Error`) est la borne pour les variables qui sont à annuler. Le champ suivant (`Link_Method`) signale la méthode globale de résolution. `Global_NR` utilise la méthode de NEWTON-RAPHSON globalement pour un module composé. Cela veut dire que toutes les variables de toutes les interfaces d'un module composé forment un système d'équations qui est à resourdre. La méthode `Locale` traite chaque interface comme un système d'équations indépendant. Par la méthode `Async`, une nouvelle tache indépendante est lancée pour chaque interface. Le dernier champ (`Way_Of_Init`) détermine la méthode de l'initialisation des frontières qui sont communes à un module et ses sous-modules. Le string `Coherent` déclenche un contrôle de la cohérence des valeurs du module avec celles des sous-modules. `From_Global` impose les valeurs du modules aux sous-modules et `From_Local` prend les valeurs des sous-modules comme valeurs initiales du module-composé. Si Motor-2 n'arrive pas à lire le fichier `link.cfg` il met un message correspondant et il continue avec les valeurs de la figure B.9.

```
-- configuration des connexions
| 100            |-- max iterations per composite module
| 3              |-- steps when Jac is to be reevaluated (1=each iteration, 2=every 2nd)
| 1.0            |-- relaxation factor of newton correction
| 0.001          |-- max X error
| 0.001          |-- max error for Sum(Y)
|Global_NR       |-- type of connection (async, local, global )
|From_Global     |-- initialisation
|Minpack         |-- algorithm for resolution (Num_Rec | Minpack)
>----------------
```

FIG. B.9 - *Le fichier de configuration `link.cfg` contient des paramètres pour les modules composés. Ici on voit les valeurs qui sont prises par défaut si le fichier n'est pas trouvé. On remarque l'utilité des commentaires.*

Un deuxième fichier de configuration `debug.cfg` n'a pas grande utilité pour un utilisateur, mais il peut servir au développeur de classes en cas de problèmes numériques[2].

Fichiers d'installation

Les fichiers d'installation sont des fichiers indispensables pour le fonctionnement de Motor-2. Ils spécifient des différents types et classes connus et facilitent la maintenance de Motor-2. Eux aussi sont composés d'un champ par

2. Le fichier `debug.cfg` contient trois champs de valeurs booléennes pour contrôler l'affichage à l'écran de quelques résultats. Le premier champ (`Show_Jacobian`) fait afficher la matrice *Jacobienne* à chaque pas d'itération, le deuxième (`Debug_Flux`) n'est plus utilisé et le troisième (`Show_Hashing`) affiche les différentes « *hashtables* » (tables de hashage) internes.

ligne. Ils contiennent les noms des types ou classes que Motor-2 doit connaître. Les fichiers nécessaires sont `pins.list` pour les types des *pattes* (par exemple `Temperature`), `frontiers.list` pour les types des frontières (p.e. `First_Kind`), `interfaces.list` pour les types d'interfaces (`Perfect_Contact`), et le plus important `contents.list` qui contient les noms de classes connues (fig. B.10).

```
--  -*- Symbol -*-
-----------------------------------------------------------------
--  some types for link up
-----------------------------------------------------------------
| LinkUp        |
| LinkStruct    |
-----------------------------------------------------------------
-- some standard classes
-----------------------------------------------------------------
| Radiation     |
| FileData      |
| Fd1D1K        |
| Coef          |
| Cell          |
| Math_Ftn      |
-----------------------------------------------------------------
-- modal objects
-----------------------------------------------------------------
| Modal         |
>-----------------------------------------------------------------
```

FIG. B.10 - *Une partie du fichier d'installation* `contents.list`.

Fichier journal

L'utilisateur peut demander à Motor-2 qu'il écrive un fichier journal de la simulation. Faute d'un autre nom spécifié, Motor-2 crée le fichier `symbol.log`.

Comme entête, le fichier journal contient l'heure exacte de sa création, ensuite les valeurs des paramètres pour les modules composés et aussi le nom du module principal. Quand le fichier est fermé, il y est inscrit l'heure l'heure de fermeture du fichier. Toutes ces information ont le caractère d'un commentaire, c'est à dire qu'elles sont précédées d'un double tiret (fig. B.11).

À chaque fin d'un pas de temps, Motor-2 écrit dans le journal une fin de bloc ('>') et la valeur de la variable `Time` (temps simulé). Suivent les valeurs des variables d'état de tous les modules et le nombre de leurs appels. Si l'utilisateur a spécifié un journal plus détaillé (avec l'argument `-l2`), Motor-2 écrit aussi les états des module à chaque pas d'itération.

Avec les marques de blocs, on obtient un fichier structuré comme les autres fichiers SYMBOL excepté qu'il n'y a pas la structure des champs. On peut le traiter par différents utilitaires (comme par exemple perl) pour récupérer et transformer les valeurs intéressantes en vue d'un post-traitement (affichage graphique).

```
-- Hi Emacs, read this file in -*- Symbol -*- mode
--
-- logfile of simulation started at:
-- MONDAY    26 OCTOBER   1992 14:33:04
--
-- current configuration
-- Max_Passes      :     50
-- Eps_Intensive   :  1.000000E-04
-- Eps_Extensive   :  1.000000E-04
-- Link_Method     : GLOBAL_NR
-- Way_Of_Init     : FROM_GLOBAL

--
-- main unit: West_L
--
-- in module west_l :  was called #     1

>-------------------------------------------------------------------------------
--                                results of time step of T = 300.0
-------------------------------------------------------------------------------

-- State of west_l :
 0.046125 18.496281 18.917999

-- State of coef_ext :
 0.046125  0.000000

-- State of iso :
18.496281  0.046125

-- State of beton :
18.917999 18.496281

-- State of coef_int :
19.000000 18.917999

--
-- logfile of simulation closed at:
-- MONDAY    26 OCTOBER   1992 14:33:04
--
```

FIG. B.11 - *Quelques parties d'un fichier journal* `symbol.log`.

B.2 Manuel du développeur

B.2.1 Présentation générale

Un nouveau modèle développé doit être fait connu dans la bibliothèque du logiciel de simulation Motor-2. À cette fin, le modèle doit exister sous forme d'un code source qui est ensuite compilé et inclu dans le programme exécutable.

Les fonctionalités de Motor-2 – surtout la simultanéité des calculs – imposent quelques contraintes sur la structure du code source. Ces contraintes sont essentiellement dues à une description cohérente de tous les modèles. Chaque instance d'un modèle doit être capable de lire ses propres paramètres et de répondre aux appels dynamiques lors de la simulation. Pour faciliter l'écriture structurée des modèles, un format spécial de fichier a été développé, le mdl. Ce format contient des champs qui sont à remplir par un développeur de modèle. Ensuite, un logiciel utilitaire lit un fichier ayant ce format et génère le code source et les intructions nécessaires pour recréer un nouvel exécutable Motor-2 connaissant le nouveau modèle.

Le manuel de l'utilisateur se divise donc en deux partie. La première décrit le format du fichier mdl, la deuxième explique l'organisation des fichiers et commandes qui font connaître un nouveau modèle à Motor-2.

B.2.2 Description de modèle, le format mdl

Les modèles de simulation que Motor-2 peut utiliser sont décrits dans des fichiers du format mdl. Ils ont donc l'extension `.mdl`. De plus, le nom du fichier doit correspondre au nom du modèle. Si le modèle s'appele coef, le fichier qui contient la description de ce modèle s'appele `coef.mdl`. Cette égalité entre le nom de fichier et le nom de modèle est nécessaire pour le mécanisme d'héritage réalisé dans la bibliothèque de Motor-2. Chaque nouveau modèle doit nommer un modèle déjà existant. Il en hérite les fonctions s'il n'en spécifie pas de nouvelles.

De même que dans les fichiers des modules, le fichier mdl contient au début une partie gestionnaire dont le format est le même que pour les fichiers smd. Elle est composée de quatre champs. Le premier contient le numéro de version de la structure mdl, généralement `0.2`. Le deuxième champ contient le nom du modèle, le troisième le nom du modèle dont le modèle actuel hérite les fonctions. Un quatrième champ indique la version du modèle.

Dans ce qui suit, la structure générale est différente de celle des fichiers de modules. Comme le contenu du fichier est à inclure dans un code de programme, la structure est très liée au langage Ada. Mais elle est toujours organisée en champs. Un champ commence par un mot clé dans une ligne comme `CODE.INIT` par exemple. Il est terminé par une ligne qui commence par un '>'. Toutes les lignes entre le mot clé et et la ligne de terminaison doivent obéir à la syntaxe de Ada, car elles sont copiées sans modifications dans un fichier source soumis

```
--                          -*- Mode: Symbol -*-
-------------------------------------------------------------------------------
| 0.1       | struct version  |
| coef      | modele name     |
| general   | origin          |
| 1.1       | modele version  |
>-------------------------------------------------------------------------------
```

FIG. B.12 - *L'entête d'un fichier mdl.*

au compilateur. L'ordre des champs n'influence pas les fichiers résultant. Pour faciliter la maintenance on essayera de respecter l'ordre indiqué ici.

Un modèle au sens de la bibliothèque Motor-2 consiste en trois parties: les paramètres libres, les routines pour lire et écrire les paramètres dans les fichiers des modules. La partie principale contient les actions dynamiques avec la définition des variables locales.

Paramètres libres

La définition des paramètres se fait par les trois mots-clés suivant.

`PARAMETER.DEPENDENCE` Ce champ reste généralement libre. Il sert à inclure des modules par des clauses `with`, dont on a besoin pour des types de paramètre très compliqués. Le modèle MODAL contient ici la ligne `with Complex_Matrix;`

`PARAMETER.LOCAL` Le plus souvent, ce champ peut également rester libre. Si d'autres modules ont été inclus dans le champ précédent, ils doivent être rendus visibles ici par une instruction `use`. D'autres déclarations locales sont possibles. Voyons encore le modèle MODAL dans la figure B.13.

```
PARAMETER.LOCAL
Modal_Version : constant String := "5.0"; -- MAL version
NMax_Node     : constant Natural := 10_000;
subtype State_Index is Natural range 1..NMax_Node;
subtype Node_Index  is Natural range 1..NMax_Node;
type T_Nature is (Real, Cplx);
>
```

FIG. B.13 - *Des définitions locales du modèle* MODAL.

`PARAMETER.DEFINITION` Chaque ligne contient le nom du paramètre et son type séparé par ':'. Elle est terminée par un ';'.

Les instructions données par le développeur dans ces trois champs sont ensuite combinées avec le squelette pour constituer un « *paquetage* » valide en Ada.

```
PARAMETER.DEFINITION
 Out_File_Name   : Vstring;
 Show_Time_Value : Boolean;
 Print_Header    : Boolean;
 At_Each_Call    : Boolean;
>
```

FIG. B.14 - *La déclaration des paramètres du modèle* PRINTER.

```
--                                -*- Mode: Ada -*-
-- <M>.a ---

with Numeric;
with Real_Matrix;
with Info_Type;
PARAMETER.DEPENDENCE

package <M>_Param is

  use Numeric;
  use Real_Matrix;
  use Info_Type;
PARAMETER.LOCAL

  type t_<M> is record
    Info : T_Info;
PARAMETER.DEFINITION
  end record;

  type p_<M> is access t_<M>;

end <M>_Param;

package body <M>_Param is
end <M>_Param;
```

FIG. B.15 - *Le fichier de squelette de la définition des paramètres.*

La suite des charatères "<M>" sera remplacée par le nom du modèle. Les mots-clés seront remplacés par lignes de code que le développeur a mises dans le fichier mdl.

Entrées/sorties

Le programme Motor-2 doit être capable de lire les paramètres dans un fichier de module. Pour tous les type de base, Motor-2 connaît les méthodes de lecture. Si les paramètres d'un modèle sont uniquement du type entier, virgule-flottant, booléen, des vecteur d'entiers ou de réels, il n'est pas néccessaire de programmer des procédures de lecture. Le mécanisme de génération de code sait les traiter correctement. Néanmoins, pour des types de paramètres compliqués, si le développeur le souhaite, il peut simplement indiquer son propre format de

lecture à l'aide des mots-clés suivant :

IO.DEPENDENCE Pour utiliser des *paquetages* qui existent ailleurs dans la bibliothèque Ada, la clause `with` permet de les inclure.

```
with Generic_Symbol_IO;
```

IO.LOCAL_DEC Dans cette partie, le développeur peut écrire ses routines utilitaires. S'il utilise des types énumérés, une instance du paquetage `Generic_Symbol_IO` permet une lecture en style SYMBOL.

```
package Nature_IO is new Generic_Symbol_IO (T_Nature);
```

IO.GET La réalisation de la procédure de lecture. La déclaration des variables locales est séparée du code même par le mot `begin`. Les paramètres de type standard peuvent être lus par l'instruction `Get_Symbol (File, Item.Var, Stop);` où `Item.Var` est le nom du paramètre. Quelquefois, le nombre des paramètres dépend du nombre des variables de sortie du module, si, par exemple, l'utilisateur veut spécifier un facteur de pondération pour chacune des sorties. Par la variable `B`, le programmeur a accès au *bloc*, donc au nombre d'entrées et de sorties de ce modules (voir figure B.16).

```
IO.GET
 Eps : P_Lvector;
 V   : P_Matrix;
begin
 Eps := new Lvector (B.Input'RANGE);
 V   := new Matrix (B.Input'RANGE, B.Input'RANGE);
 Get (File, Eps.all);
 ...
```

FIG. B.16 - *Une procédure de lecture avec accès à la variable B.*

IO.PUT La réalisation de la procédure d'écriture. De la même manière que pour la procédure de lecture il faut séparer les déclarations locales du code par le mot `begin`.

Ces champs (`IO.*`) sont copiés dans la squelette d'entrée/sortie de la figure B.17.

Code de simulation

La partie la plus importante d'un modèle concerne son comportement dynamique, c'est à dire les actions à effectuer lors de la simulation. Dans cette section, nous trouvons 14 mots clés.

CODE.TASKSIZE Un chiffre entier pour indiquer la taille de mémoire pile statique (*static stack memory*) réservée en mémoire pour chaque instance d'un module. Sa valeur minimale est de 16 kO. Ce champ peut être omis.

```
--                                -*- Mode: Ada -*-
-- <M>_io_b.a ---

with Text_IO;
with IO_Globals;
with Schema;
with <M>_Param;
IO.DEPENDENCE

package body <M>_IO is
  use  Text_IO;
  use  Schema;
  use  <M>_Param;
IO.LOCAL_DEC

  procedure Get (File : in File_Type; Item : out t_<M>; B : in p_Block) is
    use IO_Globals;
    Stop : Boolean;
IO.GET
  end Get;

  procedure Put (File : in File_Type; Item : in  t_<M>) is
    use  IO_Globals;
IO.PUT
    Put_Line (File, Stop_Line);
  end Put;

end <M>_IO;
```

FIG. B.17 - *Le fichier principal de squelette pour les entrées/sorties.*

`CODE.DEPENDENCE` Ici on peut inclure des paquetages qui existent dans la bibliothèque Ada. La ligne suivante est habituelle pour utiliser les fonction mathématique de base.

```
with Mathematical_Functions;
```

`CODE.LOCAL_DEC` Les déclaration locales que l'on peut spécifier dans ce champ comprennent les clauses `use` et toutes les variables locales, c'est à dire les variables nécessaires pour la communication et les variables propres aux calculs comme l'état interne du modèle.

Plus particulièrement, on a besoin pour la communication des vecteurs `Curr_In` et `Curr_Out` qui contiennent les valeurs des sous-vecteurs d'entrée ou de sortie respectivement de toutes les frontières. Habituellement la variable `Theta` (également un vecteur) contient l'état. Dans la figure B.18 ces variable sont marquées par le commentaire `-- standard block`.

Si le nombre d'entrées et de sorties est constant, il peut être utile de renommer les éléments des vecteurs pour faciliter l'écriture des formules dans les autres champs. Les paramètres lus dans le fichier `smd` sont disponibles dans la variable `Params`.

`CODE.INIT` Le code de ce champ est exécuté une seule fois, lors de l'initialisation du module. La variables `Curr_In` est à initialiser dans cette partie. La figure B.19 montre un exemple.

```
CODE.LOCAL_DEC
  use Mathematical_Functions;
  ------------------------ standard block
  M_In_Len        : constant := 2*2;
  M_Out_Len       : constant := 3*2;
  Curr_In         : Cvector(1.. M_In_Len);
  Curr_Out        : Cvector(1.. M_Out_Len);
  Theta           : Cvector(1.. M_In_Len+M_Out_Len+2);
  ------------------------ standard block
  R   : constant := Physical_Constants.Gas_Constant;
  G   : constant := Physical_Constants.Gravity_Acceleration;
  M   : constant := Physical_Constants.Molar_Mass_Of_Air;
  Atm : constant := 101325.0;
  T_0 : constant := 273.15;

  T_Left          : Real renames Curr_In(1);
  T_Right         : Real renames Curr_In(3);
  P_Left          : Real renames Curr_In(2);
  P_Right         : Real renames Curr_In(4);
  Phi_Left        : Real renames Curr_Out(1);
  Phi_Right       : Real renames Curr_Out(4);
  M_Dot_Left_I    : Real renames Curr_Out(2);
  M_Dot_Right_I   : Real renames Curr_Out(5);
  M_Dot_Left_S    : Real renames Curr_Out(3);
  M_Dot_Right_S   : Real renames Curr_Out(6);

  Z               : Real;
  Egal            : Boolean;
  Delta_Rho       : Real;
  Delta_P         : Real;
  Rho_Left        : Real;
  Rho_Right       : Real;

  D_Y             : Real renames Params.Dimension(2);
  D_Z             : Real renames Params.Dimension(3);
  Cp              : Real renames Params.Capa_P;
  N               : Real renames Params.N;
  K               : Real renames Params.K;
>
```

FIG. B.18 - *Les déclarations locales du modèle* CONV_HORIZ.

```
CODE.INIT
  -- initialize input
  for I in Curr_In'RANGE loop
    Curr_In (I) := Me.Block.Input(I).Val;
  end loop;
  -- open the output file
  S_Create (Prn, Str (Params.Out_File_Name));
>
```

FIG. B.19 - *La partie initialisation du modèle* PRINTER.

`CODE.SYNC_INIT_IN` Ce champ n'est que rarement utilisé. Il sert à recopier les valeurs des frontières lors d'un rendez-vous entre deux tâches. Il suffit de mettre l'instruction :

```
Int_V := Curr_In;.
```

CODE.SYNC_INPUT Ce champ est également à l'intérieur d'un rendez-vous. La seule instruction à mettre est :

```
Curr_In := Int_V;
```

CODE.SYNC_NEW_INPUT C'est le champ principal où sont placés les calculs. C'est donc l'action qui est effectuée après avoir reçu de nouvelles valeurs d'entrée. Le modèle calcule les réponses dans la variable `Curr_Out`. Son nouvel état interne peut être déterminé également, mais il n'est pas encore mis à la disposition pour les calculs suivant (voir champ `NEW_TIMESTEP`).

```
CODE.SYNC_NEW_INPUT
  for I in State'Range loop
    State(I) := 0.0;
    for J in Params.Breal'Range(2) loop
      W_ij := - 1.0 / Delta_T / Params.Freal(I) *
        (1.0 - Exp (Params.Freal(I) * Delta_T)) * Params.Breal(I,J);
      State(I) := State(I) + W_ij * (Curr_In(J) - Ant_In(J));
    end loop;
    State(I) := State(I) + Ant_State(I) * Exp (Params.Freal(I) * Delta_T);
  end loop;
  -- calcul des sorties
  Curr_Out := (Params.Hreal.all * State.all) + Params.S.all * Curr_In;
>
```

FIG. B.20 - *Les calculs du modèle* MODAL.

CODE.SYNC_RESULT Dans ce champ, les valeurs de `Curr_Out` sont passées à l'autre tâche du rendez-vous :

```
Ext_V := Curr_Out;
```

CODE.AS_INIT_IN Pendant un rendez-vous en mode **async**, la valeur initiale d'une frontière est passée :

```
In_Val(1) := Curr_In(Phi.In_Pins(1).Index);
```

CODE.AS_NEW_INPUT Accepte des nouvelles valeurs d'entrée sur une frontière :

```
Curr_In(Phi.In_Pins(1).Index) := In_Val(1);
```

CODE.AS_OUT_FRONTIER Passe la valeur de résultat dans une frontière :

```
Out_Val(1) := Curr_Out(Phi.Out_Pins(1).Index);
```

CODE.NEW_TIMESTEP Au passage à un nouvel instant de la simulation, tous les modules sont avertis par cette entrée. L'état des derniers calculs peut être accepté comme nouvelle valeur. Les modèles sans état peuvent omettre ce champ.

CODE.DO_TERMINATE Si un module est forcé d'interrompre son existence, les instructions de ce champ sont exécutées. Généralement cette entrée reste libre.

```
CODE.NEW_TIMESTEP
  Theta := Theta_Old;
>
```

FIG. B.21 - *Instruction typique pour passer au nouvel état.*

CODE.FINALIZE Correspondant à l'initialisation des actions de terminaison. Le modèle PRINTER ferme à cet instant le fichier dans lequel il a écrit ses valeurs.

```
S_Close(Prn);
```

CODE.PRINT Pour des traveaux de mis au point de la simulation, on est parfois amené à suivre au pas d'itération près l'évolution des variables. Ce champ livre les routines nécessaires d'une sortie formattée. Le code généré par défaut est suffisant dans la plupart des cas. Cependant, si des variables intermédiaires qui ne sont pas parmi les entrées, les sorties et l'état, doivent être suivies, le programmeur peut mettre ici son propre code.

```
CODE.PRINT
  Module_Log.Show_State(My_Name, Theta);
  for I in Me.Block.Input'Range loop
    Write_Log (My_Name, Str (Me.Block.Input(I).Name), Curr_In(I));
  end loop;
  for I in Me.Block.Output'Range loop
    Write_Log (My_Name, Str (Me.Block.Output(I).Name), Curr_Out(I));
  end loop;
>
```

FIG. B.22 - *Les instructions par defaut pour imprimer les variables dans le fichier de journal.*

Les champs CODE.SYNC_INIT_IN, CODE.SYNC_INPUT, CODE.SYNC_RESULT, CODE.AS_INIT_IN, CODE.AS_INPUT et CODE.AS_OUT_FRONTIER sont des survivants d'une version précedente de Motor-2. Il s'est avéré qu'ils ne changent jamais, si l'on utilise toujours les mêmes noms pour les variables de communication Curr_In et Curr_Out. Leur contenu peut être fixe. De cette manière, on facilite l'écriture du fichier mdl. Les valeurs ici indiquées sont celles qui sont mises par défaut. Un développeur de modèle n'a plus besoin de s'en occuper.

Comme pour les paramètres et les entrées/sorties, les champs du fichier mdl concernant la partie CODE sont combinés avec un fichier squelette pour construire le code source du modèle. Les figures B.23, B.24 et B.25 montrent ce fichier squelette.

```
--                                -*- Mode: Ada -*-
-- This file <M>_sim_b.a was generated automatically by a Makefile.
-- Do not edit.
-- Change "gen_sim_b.skl" or "<M>.mdl" and start 'make' again.

with Numeric;
with Real_Matrix;
with VStr20;
with Module;
-------------------
--with Options;
with Verbose;
with Blinking;
with Module_Log;
with Make_Full_Name;
with Init_Done_Msg;
with Statistics;
with <M>_Param;
use  <M>_Param;
with <M>_Symbol_IO;

CODE.DEPENDENCE

package body <M>_Sim_Module is

  use Blinking;

  Debug : constant Boolean := False;


  task body <M>_Module_Type is
    use Numeric;
    use Real_Matrix;
    use VStr20;
    use Verbose;
    use Module_Log;

    Me       : Module.p_Module;
    Nr_Step_Calls : Natural := 0;
    Nr_Glob_Calls : Natural := 0;
    Params   : p_<M>;

  begin -- main task

    accept Initialize   (Myself : in  Module.p_Module) do
      Me := Myself;
    end Initialize;
    Blink(Me.My_Id, Init_Alive);

    Params := <M>_Symbol_IO.Uni_p_<M>(Me.Contents.Contents);

    Local : declare

      My_Name : constant String := Make_Full_Name(Me);

      -- Theta has to be declared.
CODE.LOCAL_DEC
```

FIG. B.23 - *Le fichier squelette (part 1) pour les actions dynamiques d'un modèle.*

```
   begin
CODE.INIT
     if Verbose.Level > Verbose.Verbose then
       Init_Done_Msg(Module_Name => My_Name,
                     Type_Name   => "<M>");
     end if;

     Blink(Me.My_Id, Stop_Init);

     loop
       select
         accept AS_Init_In (Phi    : in P_Frontier;
                            In_Val : out AS_Message_Vec) do
           null; -- if no code inserted, at least a valid null statement
CODE.AS_INIT_IN
         end AS_Init_In;
       or
         accept AS_In_Frontier (Phi    : in P_Frontier;
                                In_Val : in AS_Message_Vec) do
           null; -- if no code inserted, at least a valid null statement
CODE.AS_IN_FRONTIER
         end AS_In_Frontier;
         Blink(Me.My_Id, Alive);
         Nr_Step_Calls := Nr_Step_Calls + 1;
CODE.AS_NEW_INPUT
         Blink(Me.My_Id, Sleep);
       or
         accept AS_Out_Frontier (Phi     : in P_Frontier;
                                 Out_Val : out AS_Message_Vec) do
           null; -- if no code inserted, at least a valid null statement
CODE.AS_OUT_FRONTIER
         end AS_Out_Frontier;
       or
         accept Sync_Init_In (Int_V   : out Sync_Message_Vec) do
           null; -- if no code inserted, at least a valid null statement
CODE.SYNC_INIT_IN
         end Sync_Init_In;
       or
         accept Sync_Input    (Int_V   : in  Sync_Message_Vec) do
           null; -- if no code inserted, at least a valid null statement
CODE.SYNC_INPUT
         end Sync_Input;
         Blink(Me.My_Id, Alive);
         Nr_Step_Calls := Nr_Step_Calls + 1;
CODE.SYNC_NEW_INPUT
         Blink(Me.My_Id, Sleep);
       or
         accept Sync_Result          (Ext_V   : out Sync_Message_Vec) do
           null; -- if no code inserted, at least a valid null statement
CODE.SYNC_RESULT
         end  Sync_Result;
       or
```

FIG. B.24 - *Le fichier squelette (part 2) pour les actions dynamiques d'un modèle.*

```
          accept Fix_The_State do    -- fix theta
            Blink(Me.My_Id, Fix_Alive);
            if Module_Log.Log >= Detailed then
              Module_Log.Write_Log(My_Name, "was called #", Nr_Step_Calls);
            end if;
          end Fix_The_State;
          Nr_Glob_Calls := Nr_Glob_Calls + Nr_Step_Calls;
          Nr_Step_Calls := 0;
CODE.NEW_TIMESTEP
          Blink(Me.My_Id, Stop_Fix);
        or
          accept Show;
          null; -- if no code inserted, at least a valid null statement
CODE.PRINT
        or
          accept Do_Terminate;
          case Statistics.Print_Call_Statistics is
            when Statistics.Global_Nr_Of_Calls =>
              Statistics.Put(My_Name, Nr_Glob_Calls);
            when others =>
              null;
          end case;

          exit;
        or
          terminate;
        end select;
      end loop;
CODE.FINALIZE
    end Local;
  end <M>_Module_Type;
end <M>_Sim_Module;
```

FIG. B.25 - *Le fichier squelette pour les actions dynamiques d'un modèle.*

B.2.3 Ajouter un nouveau modèle à Motor-2

Organisation des fichiers

Les fichier mdl décrits en haut se trouvent tous dans un répertoire appellé `MDLDIR`. Ce répertoire constitue la bibliothèque de modèles pour Motor-2. Le fichier mdl est la base pour la création des fichiers source. Pour effectuer ce travail, il existe un `Makefile` qui utilise quelques utilitaires de base d'UNIX et son propre utilitaire `fill`. Des fichiers « *squelette* » sont également nécessaires. Typiquement ces fichiers se trouvent dans le `TOOLDIR`. Les fichiers source générés sont mis dans le répertoire `MDLADADIR`. Trois autres fichiers source sont modifiés par la génération d'un nouveau modèle. Il doivent se trouver dans `M2_SRCDIR`. Nous avons donc une structure de répertoires comme dans la figure B.26.

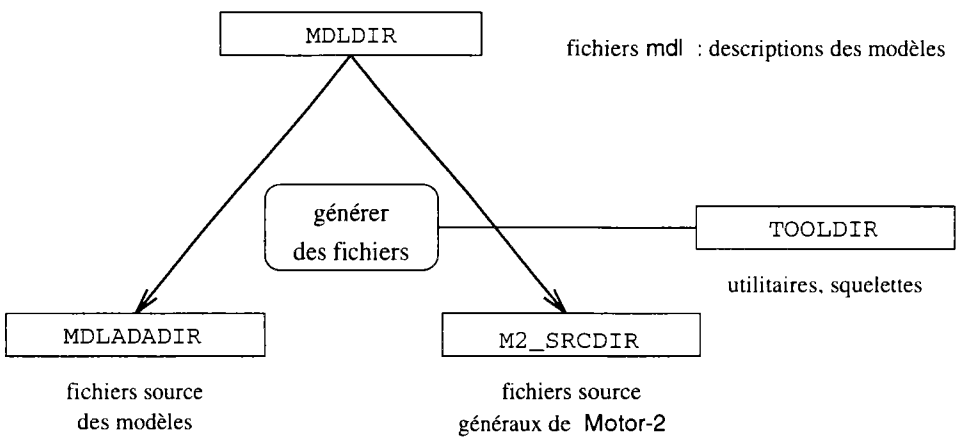


FIG. B.26 - *La structure des répertoires pour l'insertion d'un nouveau modèle dans Motor-2.*

Démarches de l'installation

Plusieurs étapes sont à effectuer pour l'installation d'un nouveau modèle :

1° remplir le fichier mdl et le mettre dans le répertoire `MDLDIR`,

2° se placer dans le `TOOLDIR`. Le nom du nouveau modèle est à mettre dans la liste de modèles au début du fichier `Makefile`. Ensuite lancer la commande `make`. Cela crée des fichiers source dans le `MDLADADIR`. Après la génération des fichiers pour les modèles, les trois fichiers suivant sont mis à jour :

- `contents_types.a`,
- `contents_io_b.a`,
- `start_module_b.a`.

Ces trois fichiers se trouvent dans le `M2_SRCDIR`.

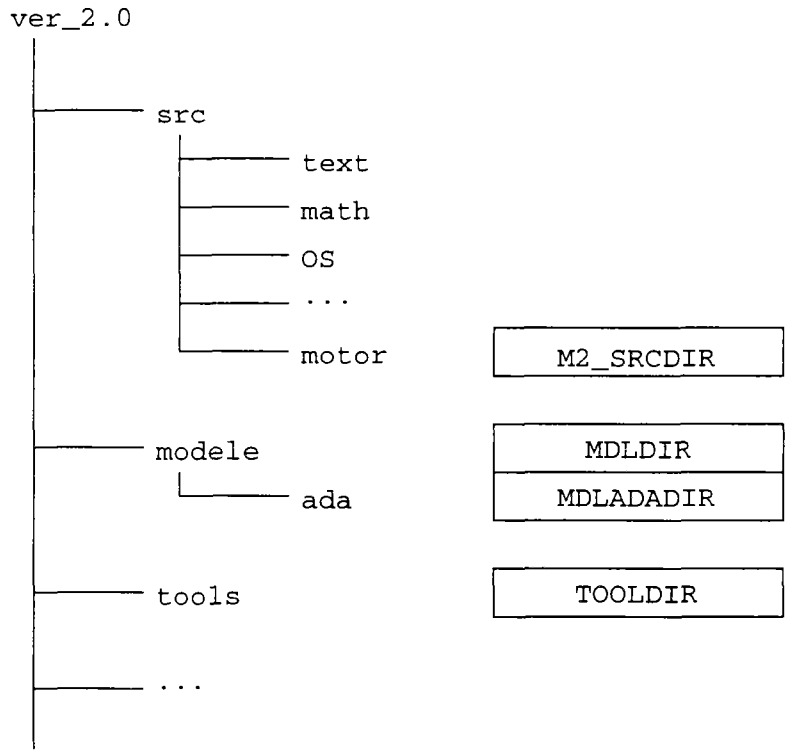


FIG. B.27 - *L'organisation momentanée des répertoires.*

3° Les fichiers créés dans `MDLADADIR` et les fichiers modifiés dans `M2_SRCDIR` doivent être recompilés. Une édition des liens nous crée un nouvel exécutable Motor-2 contenant les nouveaux modèles.

La compilation des fichiers générés et modifiés du troisième point peut s'effectuer de deux manières. D'une part, la plupart des compilateurs Ada connaissent un mode dans lequel ils détectent automatiquement tous les fichiers à recompiler. Mais parfois cela peut prendre beaucoup de temps. Une autre méthode consiste à faire générer en même temps que les fichiers source un fichier de commande pour la recompilation. La commande '`make`' du deuxième pas fournit les deux fichiers `rec.inv` et `new.inv` à cet objectif.

Annexe C

Proformas

Les fiches techniques des modèles utilisés dans cette thèse suivent le schéma suivant

- nom complet du modèle.
- nom de modèle dans la bibliothèque Motor-2.
- nombre et types des variables d'entrée/sortie et le nombre et types des frontières. La mention « *ill.* » indique que le nombre n'est pas spécifié par le modèle même. Une instance (le module) peut choisir le nombre qui convient.
- description des paramètres libres du modèle.
- équations constitutionelles et leur application.
- commentaires.

C.1 Coefficient d'échange

nom complet	Coefficient d'échange
nom Motor-2	`coef`
connexions	entrées 2 température $\boldsymbol{T}$ sorties 2 flux de chaleur ϕ frontières 2 1^{re} espèce
paramètres	coefficient d'échange : réel
équations	$$\phi_1 = h(T_2 - T_1)$$ $$\phi_2 = h(T_1 - T_2)$$ Ce modèle n'a pas d'état. Le vecteur d'entrée `Curr_In` contient T_1 et T_2. Les sorties ϕ_1 et ϕ_2 sont calculées dans `Curr_Out`.
commentaires	

C.2 Différences finies 1D

nom complet	Différences finies 1D
nom Motor-2	`fd1d1k`
connexions	entrées 2 température $\boldsymbol{T}$ sorties 2 flux de chaleur ϕ frontières 2 1re espèce
paramètres	nombre de nœuds n : entier masse volumique ρ : réel capacité calorifique c : réel conductivité λ : réel profondeur l : réel surface A : réel
équations	$$\Theta_{i+1} = \Theta_i + \Delta t(\boldsymbol{A}\Theta_i + \boldsymbol{E}\boldsymbol{T}_i)$$ $$\varphi_{i+1} = \boldsymbol{J}\Theta_{i+1} + \boldsymbol{G}\boldsymbol{T}_i$$ La profondeur de la paroi est divisée en n couches. Les températures des couches sont désignées par Θ. Les températures imposées aux bords sont $\boldsymbol{T}$ dans `Curr_in`. Les paramètres thermo-physiques sont utilisés pour constituer les matrices $\boldsymbol{A}$, $\boldsymbol{E}$, $\boldsymbol{J}$ et $\boldsymbol{G}$. Dans la deuxième équation, on ne dispose que de la valeur $\boldsymbol{T}_i$. Nous ne pouvons donc pas employer $\boldsymbol{T}_{i+1}$ qui serait plus correcte pour calculer les sorties φ dans `Curr_Out`.
commentaires	

C.3 Nœud capacitif

nom complet	Nœud capacitif
nom Motor-2	`CN`
connexions	entrées 1 température T sorties 1 flux de chaleur ϕ frontières 1 1re espèce
paramètres	capacité : réel
équations	$$\phi = C\dot{T}$$ Une simple intégration détermine le flux en fonction de la température T dans `Curr_In`. La sorties ϕ est calculée dans `Curr_Out`.
commentaires	

C.4 Modal

nom complet	Modal
nom Motor-2	`modal`
connexions	entrées ill. températures ou flux de chaleur sorties ill. températures ou flux de chaleur frontières ill.
paramètres	nombre total de nœuds : entier nombre d'états : entier nombre de nœuds : entier S : matrice réelle Sp : matrice réelle So : matrice réelle H : matrice réelle F : matrice réelle B : matrice réelle P : matrice réelle O : matrice réelle
équations	$$\dot{\boldsymbol{x}}(t) = \boldsymbol{F}\boldsymbol{x}(t) + \boldsymbol{B}\dot{\boldsymbol{U}}(t)$$ $$\boldsymbol{y}(t) = \boldsymbol{H}\boldsymbol{x}(t) + \boldsymbol{S}\boldsymbol{U}(t)$$ La variable $\boldsymbol{x}$ designe l'état interne `Theta` du modèle qui est calculé par la première des équations. Le vecteur des entrée `Curr_In` est designé par $\boldsymbol{U}$. Sa dérivée est déterminée numériquement par rapport au pas précedent. Les sorties `Curr_Out` sont calculées dans $\boldsymbol{y}$ par la deuxième équation. La matrice diagonale $\boldsymbol{F}$ est la « *matrice des valeurs propres* », $\boldsymbol{B}$ est la « *matrice de commande* », $\boldsymbol{H}$ est la « *matrice de sortie* », et $\boldsymbol{S}$ est appelée la « *matrice statique* ».
commentaires	pour plus d'information sur la méthode modale, voir [52].

C.5 Radiosité

nom complet	Radiosité
nom Motor-2	`radiosity`
connexions	entrées ill. températures sur les surfaces $\boldsymbol{T}$ sorties ill. radiosités des surfaces $\boldsymbol{B}$ frontières ill.
paramètres	emissivité ε : vecteur réel reflectivité ρ : vecteur réel facteurs de forme $\boldsymbol{F}$: matrice réelle
équations	$$\boldsymbol{B} = \sigma\varepsilon(\boldsymbol{I} - \rho\boldsymbol{F})^{-1} \cdot \boldsymbol{T}^4$$ Calcule dans $\boldsymbol{B}$ `Curr_Out` les radiosités des surfaces en fonction des températures dans `Curr_In`.
commentaires	

C.6 Éclairage des cubes

nom complet	Éclairage des cubes
nom Motor-2	`cube_ecl`
connexions	entrées ill. radiosités de l'enceinte $\boldsymbol{B}$ sorties 6 éclairage des surfaces $\boldsymbol{E}$ frontières ill. + 6
paramètres	facteurs de forme $\boldsymbol{F}$: matrice réelle
équations	$$\boldsymbol{E} = \boldsymbol{F} \cdot \boldsymbol{B}$$ Calcule l'éclairage de ses propres six surfaces dans `Curr_Out` en fonction des radiosités `Curr_In` vues sur les surfaces de l'enceinte.
commentaires	

C.7 Convection simplifiée horizontale

nom complet	Convection simplifiée horizontale
nom Motor-2	`conv_horiz`
connexions	entrées 4 : 2 × température T 2 × pression p sorties 6 : 2 × débit supérieur $\dot{m}_s$ 2 × débit inférieur $\dot{m}_i$ 2 × flux de chaleur ϕ frontières 2 : convection horizontale
paramètres	dimension : vecteur 3D réel capacité spec. : réel n : réel k : réel
équations	$\rho = \frac{p}{RT}$ $z_n = \frac{p_1 - p_2}{(\rho_1 - \rho_2)g}$ $\dot{m}_s = kl\rho(\Delta\rho g)^n \frac{(h - z_n)^{n+1}}{n+1}$ $\dot{m}_i = kl\rho(\Delta\rho g)^n \frac{(z_n)^{n+1}}{n+1}$ $\phi = \dot{m}_s c_{p1} T_1 - \dot{m}_i c_{p2} T_2$ Avec les températures et les pressions dans `Curr_In`, les équations internes sont appliqués. Les résultats `Curr_Out` contiennent les débits massiques et les flux de chaleur. l contient la largeur de l'ouverture, n est une valeur empirique décrivant le type d'écoulement, k représente la perméabilité. Voir § 8.3.2 pour la modélisation d'un échange convectif simplifié.
commentaires	

C.8 Convection simplifiée verticale

nom complet	Convection simplifiée verticale
nom Motor-2	`conv_vert`
connexions	entrées 4 : 2 × température T 2 × pression p sorties 2 : 2 × débit massique $\dot{m}$ 2 × flux de chaleur ϕ frontières 2 : convection verticale
paramètres	dimension : vecteur 3D réel capacité spec. : réel n : réel k : réel
équations	$\dot{m} = k\rho S(p - p_t)^n$ $\phi = \dot{m}c_pT$ Avec les températures et les pressions dans `Curr_In`, les équation internes sont appliquées. Les résultats `Curr_Out` contiennent les débits massiques et les flux de chaleur. l contient la largeur de l'ouverture, n est une valeur empirique décrivant le type d'écoulement, k représente la perméabilité. Voir § 8.3.2 pour la modélisation d'un échange convectif simplifié.
commentaires	

C.9 Régulation de chauffage

nom complet	Régulation de chauffage
nom Motor-2	`z_ctrl`
connexions	entrées 2 température T_i sorties 1 puissance P frontières 3
paramètres	puissance max. $P_{\max}$: réel température cible T_c : réel largeuer de bande T_b : réel
équations	$$T_s = \frac{T_1 + T_2}{2}$$ $$P = \frac{P_{\max}}{2}\left(1 - \tanh\frac{T_s - T_c + T_b}{T_b}\right)$$ La moyenne des températures d'entrée dans `Curr_In` est considérée comme la température de la sonde. La puissance à émettre P est calculée dans `Curr_Out`.
commentaires	

C.10 Échange circulaire forcé entre radiateur et cellules

nom complet	Échange circulaire forcé entre radiateur et cellules
nom Motor-2	`z_htrn`
connexions	entrées ill. température T_i sorties ill. flux de chaleur ϕ_i frontières ill. 1$^{\text{er}}$ espèce
paramètres	débit massique $\dot{m}$: réel capacité calorifique de l'air C_p : réel conductance h : réel
équations	$$\phi_{\text{radiateur}} = h(T_{\text{radiateur}} - \frac{T_{\text{out}} - T_{\text{in}}}{2})$$ $$\phi_i = \dot{m}C_p(T_{i-1} - T_i)$$ Dans chaque cellule, l'air est représenté par une température $T_1 \cdots T_{n-1}$. L'air sort du radiateur, traverse chaque cellule et rentre dans le radiateur. Son débit massique $\dot{m}$ est le même partout. En fontion des températures des cellules T_i et du radiateur T_n dans `Curr_In`, ce modèle calcule pour chaque élément le flux effectif (reçu - cédé) dans `Curr_Out`.
commentaires	voir aussi [85].

Annexe D

Méthodes numériques

D.1 Systèmes d'équations non-linéaires (SENL)

Nous regardons le problème de N fonctions $f_1 \dots f_N$ de N variables indépendantes $x_1..x_N$.

$$\begin{aligned} f_1(x_1, x_2, \dots, x_N) &= 0 \\ f_2(x_1, x_2, \dots, x_N) &= 0 \\ &\vdots \\ f_N(x_1, x_2, \dots, x_N) &= 0 \end{aligned}$$

ou en forme vectorielle

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0} \tag{D.1}$$

Dans le cas général les fonctions $\boldsymbol{f}$ sont non-linéaires ce qui nous empêche d'appliquer les méthodes directes de résolution.

D.1.1 La méthode standard de NEWTON-RAPHSON

L'approche de NEWTON-RAPHSON consiste en une linéarisation du problème initial. On remplace $\boldsymbol{f}(\boldsymbol{x})$ par un système de fonctions linéaires $\boldsymbol{f}^*(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}$. La solution du système linéaire $\boldsymbol{f}^*(\boldsymbol{x}) = \boldsymbol{0}$ est une approximation de la solution de $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0}$.

Pour la linéarisation de $\boldsymbol{f}(\boldsymbol{x})$ nous faisons une expansion de la série de TAYLOR,

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{x}^0) + \boldsymbol{J}_{\boldsymbol{x}^0}(\boldsymbol{x} - \boldsymbol{x}^0) + E(\boldsymbol{x}) \tag{D.2}$$

et nous nous arrêtons après le premier terme. Les termes suivants d'ordre supérieur ($E(\boldsymbol{x})$) seront négligés ; on peut les voir comme erreur de la méthode.

La matrice $\boldsymbol{J}_{\boldsymbol{x}^0}$ contient toutes les dérivées partielles de toutes les foncions f_i par rapport à toutes les variables x_j au point $\boldsymbol{x}_0$.

$$\boldsymbol{J} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \frac{\partial f_1}{\partial x_N} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \frac{\partial f_2}{\partial x_N} \\ \vdots & & & \vdots \\ \frac{\partial f_N}{\partial x_1} & \frac{\partial f_N}{\partial x_2} & \dots & \frac{\partial f_N}{\partial x_N} \end{bmatrix} \tag{D.3}$$

Habituellement on appelle la matrice J la *Jacobienne.*

Maintenant nous essayons de résoudre le système linéarisé

$$\boldsymbol{f}^*(\boldsymbol{x}) = \boldsymbol{J}_{\boldsymbol{x}^0}(\boldsymbol{x} - \boldsymbol{x}^0) + \boldsymbol{f}(\boldsymbol{x}^0) = \boldsymbol{0} \tag{D.4}$$

ce qui donne comme premier résultat (si $\boldsymbol{J}_{x^0}$ n'est pas singulière)

$$\boldsymbol{x} = \boldsymbol{x}^0 - \boldsymbol{J}_{\boldsymbol{x}^0}^{-1}\boldsymbol{f}(\boldsymbol{x}^0). \tag{D.5}$$

Comme la solution de $\boldsymbol{f}^*$ est une approximation, on applique cette linéarisation de nouveau après chaque solution obtenue. Cela nous amène au schéma d'itération

$$\boldsymbol{x}^{i+1} = \boldsymbol{x}^i - \boldsymbol{J}_{\boldsymbol{x}^i}^{-1} \cdot \boldsymbol{f}(\boldsymbol{x}^i) \tag{D.6}$$

Cette prescription est appelée la méthode de NEWTON-RAPHSON. En général, on arrête l'itération quand une norme $||\boldsymbol{x}^i - \boldsymbol{x}^{i-1}|| < \epsilon$ ou quand chaque élément du vecteur est inférieur à un ϵ_j correspondant $(\boldsymbol{x}_j^i - \boldsymbol{x}_j^{i-1}) < \epsilon_j$, où le seuil d'arrêt est différent et doit être vérifié pour chaque élément.

Modifications

On voit, qu'il faut déterminer toutes les dérivées $\boldsymbol{J}_{\boldsymbol{x}^i}$ et inverser une matrice $\boldsymbol{J}_{\boldsymbol{x}^i}^{-1}$ à chaque pas d'itération. C'est sont des efforts de calculs très importants, car l'inversion d'une matrice est une action coûteuse et la détermination numérique de dérivées par une variation d'entrée cause N^2 évaluations de fonctions. Il faut donc minimiser les calculs de la *Jacobienne.* Une simplification de la méthode standard utilise la même matrice de dérivées pour plusieurs pas d'itération consécitfs. La vitesse de convergence diminue. Où la méthode originale de NEWTON-RAPHSON est de l'ordre 2, la méthode simplifiée est de l'ordre 1. Généralement, on fait recalculer la *Jacobienne* après 3 – 7 itérations avec la même.

Pour utiliser la méthode de NEWTON-RAPHSON sur des système qui ne sont pas seulement non-linéaires mais aussi non-continus, on peut introduire un

facteur de *relaxation* $\lambda < 1$. Le pas correctif Δx est multiplié avec ce facteur pour ralentir l'approximation.

$$\boldsymbol{x}^{i+1} = \boldsymbol{x}^{i} - \lambda \cdot \boldsymbol{J}_{\boldsymbol{x}^i}^{-1} \cdot \boldsymbol{f}(\boldsymbol{x}^{i}) \tag{D.7}$$

De cette façon, on augmente la probabilité de convergence pour les équations avec hystérès ou des fonctions non-continues. Cette méthode plus d'autres astuces pour éviter de recalculer la *Jacobienne* sont implémentés dans la méthode de POWELL. Une implémentation avec des modifications supplémentaires se trouve dans le paquétage Minpack [39], [64].

D.2 Systèmes d'équations différentielles ordinaires (ODE)

La solution numérique des systèmes d'équations différentielles ordinaires est obtenue en remplaçant les éléments différentiels dx et dy par des incréments ou pas finis Δx et Δy

$$y' = \frac{dy}{dx} = f(x,y) \quad \Rightarrow \quad y(x+\Delta x) = y(x) + \Delta x \cdot \Phi(f(x,y);\Delta x) \tag{D.8}$$

ou Φ est une méthode qui dépend de la fonction $f(x,y)$, de la longueur du pas Δx et éventuellement des valeurs prises aux pas précédents. De proche en proche on obtient les nouvelles valeurs de $f(x,y)$.

On dispose de trois types de méthodes pour obtenir la solution:

- les méthodes à un pas font une ou plusieurs évaluations de la fonction derivée et combinent les résultats obtenus pour égaler le schéma au développement de TAYLOR. La méthode de RUNGE-KUTTA en est une représentante efficace.

- les méthodes d'extrapolation. On parcourt plusieurs fois un grand intervalle avec des pas de longueurs différentes, puis on extrapole les résultats pour le pas de longueur limite $h = 0$.

- les méthodes *prédicteur-correcteur* conservent les solutions obtenues aux pas précédents pour déterminer une valeur extrapolée du prochain pas qui sera ensuite améliorée par itération à l'aide d'une formule implicite.

Certains problèmes conduisent à résoudre des systèmes d'équations différentielles qui posent de grandes difficultés numériques. Ils sont appelés problèmes «raides», parce qu'il y a au moins deux valeurs propres du système qui sont très différentes. On a développé des méthodes avec inégration rétrograde qui permettent d'utiliser des pas plus grands qu'ils ne sont nécessaires pour la stabilité avec des méthodes traditionnelles.

D.2.1 Introduction

Un grand nombre d'équations différentielles sont intégrables. Mais la plupart ne le sont que numériquement. On cherche à trouver des valeurs vérifiées par une fonction dont on ne peut pas trouver une représentation analytique.

Une équation différentielle ordinaire (ODE) du premier ordre s'écrit

$$y' = f(x, y) \tag{D.9}$$

avec $y = y(x)$ et $y' = dy/dx$.

Des problèmes décrits par des ODE d'un ordre supérieur ou égal à deux peuvent toujours être ramenés au traitement des systèmes d'équations différentielles (SED) du premier ordre. Par exemple dans l'équation du second ordre

$$\frac{d^2y}{dx^2} + q(x)\frac{dy}{dx} = r(x) \tag{D.10}$$

on remplace dy/dx par z et on peut écrire

$$\begin{aligned} \frac{dy}{dx} &= z(x) \\ \frac{dz}{dx} &= r(x) - q(x)z(x) \end{aligned}$$

ce qui mène à un système de deux équations différentielles du premier ordre.

Habituellement on choisit comme nouvelle variable la dérivée d'une autre. Le problème initial d'une ODE d'ordre m est réduit à l'étude d'un système de m équations différentielles du premier ordre:

$$\begin{aligned} y_1' &= f_1(x, y_1, y_2, \ldots y_m) \\ y_2' &= f_2(x, y_1, y_2, \ldots y_m) \\ &\vdots \\ y_m' &= f_m(x, y_1, y_2, \ldots y_m) \end{aligned} \tag{D.11}$$

ou écrit sous forme vectorielle

$$\boldsymbol{y}' = \boldsymbol{f}(x, \boldsymbol{y}) \tag{D.12}$$

dont on connait les fonctions $\boldsymbol{f} = f_1..f_m$.

Un problème différentiel n'est pas entièrement spécifié par ses seules équations. Les conditions au bords déterminent la méthode numérique la plus adaptée pour le problème. Le cas qui nous intéresse ici est une condition initiale:

$$\boldsymbol{y}(x_0) = \boldsymbol{y}_0 \tag{D.13}$$

Toutes les valeurs $\boldsymbol{y}_0$ sont connues pour un point de départ x_0 et on cherche les valeurs $\boldsymbol{y}_i$ pour des points $x_i \neq x_0$. Les problèmes à condition initiale sont appelés problèmes de CAUCHY.

Les cas ou il faut trouver une équation qui passe par plusieurs points (*multi-point-problem*) ne sont pas traités ici comme par exemple:

$$w'' + w = 0,$$

avec les conditions aux bords

$$w(0) = 0, w(\pi) = 0$$

On trouve de bonnes explications relatives à des tels problèmes dans la littérature ([75] chap. 7.3)

Pour alléger l'écriture dans les chapitres suivants on ne considéra qu'une seule équation différentielle et non un SED. Pour résoudre un SED il suffit d'appliquer les mêmes méthodes pour chacune des m fonctions du système.

Unicité de la solution

Il existe une seule solution pour le problème de Cauchy ([26] chap. 3.2, [75] chap. 7.1)

$$y' = f(x, y) \tag{D.14}$$

$$y(x_0) = y_0 \tag{D.15}$$

si

- f est continue dans un intervalle autour de x_0 et
- il existe un $L \forall x, y_1, y_2 \neq y_1 \; |f(x, y_1) - f(x, y_2)| \leq L|y_1 - y_2|$ [1]

La dernière condition est surtout accomplie, si

$$\max_{x,y \in D} |\frac{\partial f}{\partial y}| = L < \infty \tag{D.16}$$

Cela veut dire que, si la fonction f n'a pas de dérivée infinie (asymptote verticale / pôle), il n'y a qu'une seule courbe qui passe par (x_0, y_0) et qui est solution de $y' = f(x, y)$ ([26, chap. 3.2], [32, chap. 10.1], [75, p.408/409]).

D.2.2 Méthode d'EULER-CAUCHY

La méthode d'EULER-CAUCHY n'est pas très puissante, mais elle permet de bien comprendre les principes fondamentaux de la résolution numérique des SED.

On choisit $N + 1$ points

$$x_0 < x_1 < ... < x_{N-1} < x_N \tag{D.17}$$

1. On appelle cette relation la *condition de* LIPSCHITZ et L est appelé la *constante de* LIPSCHITZ.

dans l'intervalle D à étudier. Les différences entre deux points sont notées:

$$h_i = x_{i+1} - x_i; \quad h = \max_i h_i; \quad \forall i \in [0, N-1]. \tag{D.18}$$

Il faut remarquer que nous utilisons dans tout le texte suivant la notation y^* pour des valeurs exactes et simplement y pour une valeur approximée.

Si on intègre l'équation D.9 réparant deux points voisins x_i et x_{i+1} on obtient

$$y^*_{i+1} - y^*_i = \int_{x_i}^{x_{i+1}} f(s, y(s))\, ds. \tag{D.19}$$

Une première approximation de l'équation D.19 est faite par

$$y_{i+1} = y_i + h_i f(x_i, y_i) \tag{D.20}$$

qui remplace simplement l'intégrale par le rectangle située au-dessous de la courbe (voir fig. D.1).

Le schéma défini par D.20 s'appelle schéma d'EULER-CAUCHY. C'est une méthode directe pour calculer une nouvelle valeur y_{i+1} en fonction de la valeur du pas précédent.

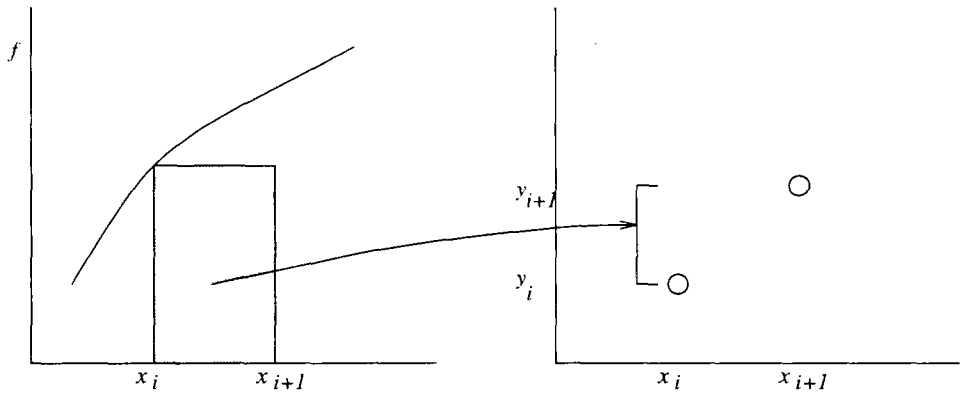


FIG. D.1 - *La méthode d'*EULER-CAUCHY.

Erreur de la méthode d'EULER-CAUCHY

Nous allons chercher une estimation de l'erreur locale

$$\varepsilon_i = y^*_i - y_i \tag{D.21}$$

entre la solution exacte (et inconnue) y^*_i et la solution approchée y_i par le schéma D.20 calculée avec une précision infinie (pas d'arrondis).

Un développement de TAYLOR de l'équation D.19 pour le pas i en supposant que y_i soit "exacte" $(y^*_i = y_i)$ mène à

$$y^*_{i+1} = \underbrace{y_i + h_i f(x_i, y_i)}_{y_{i+1}} + \underbrace{\frac{h_i^2}{2} f'(\xi_i)}_{\varepsilon_i}, \quad \xi_i \in [x_i, x_{i+1}] \tag{D.22}$$

Les deux premiers termes représentent l'approximation réellement utilisée par la méthode d'EULER-CAUCHY et le dernier terme ε_i représente l'*erreur locale* de la méthode au pas i. Cette erreur est seulement relative à un pas, x_i - x_{i+1}, et elle est apparemment d'ordre $O(h^2)$. Les erreurs dues aux pas précédents sont prises en compte par l'*erreur globale* de la méthode, notée e_N:

$$e_N(x) = y^*(x) - y_N(x). \tag{D.23}$$

Cette équation exprime la différence entre la solution exacte $y^* = y^*(x_0+H)$ et la valeur approchée $y_N(x_0 + H)$ par N pas de la méthode. Une majoration de e_N mène à l'expression [44], [75, p. 412/413], [26, p. 75/76]

$$|e_N| \leq c_N h = O(h);\ c_n = conste. > 0 \tag{D.24}$$

dont le résultat est évident

$$\lim_{h \to 0} e = 0. \tag{D.25}$$

En minimisant la longueur du pas h on minimise l'erreur que l'on commet. La méthode d'EULER-CAUCHY est donc convergente.

En général on parle d'une méthode d'ordre p, si l'erreur globale e est de la forme:

$$e = O(h^p). \tag{D.26}$$

La méthode d'Euler-Cauchy est alors d'ordre un.

Une expansion asymptotique de e que l'on suppose d'ordre p mène à une approximation (cf. [75, p. 419])

$$e(x) = e_p(x)h^p + e_{p+1}(x)h^{p+1} + \cdots \tag{D.27}$$

à laquelle nous reviendrons plus tard.

Erreur d'arrondi

A chaque pas on fait de plus des erreur d'arrondi, parce qu'on calcule en fait la solution d'un schéma D.20 perturbé

$$y^*_{i+1} = y^*_i + h_i[f(x_i, y^*_i) + \mu_i] + \rho_i \tag{D.28}$$

où μ_i désigne l'erreur de la méthode (plus l'erreur de calcul et de connaissance de la fonction f) et ρ_i est l'erreur d'arrondi [26, p. 78/79]). L'erreur d'arrondi est de l'ordre $0.5 \cdot 10^{-precision}$ où *précision* est le nombre de chiffres avec lequel les calculs sont effectués sur un ordinateur.

Si on augmente le nombre de pas N pour $x_0 + H$, on exécute plus souvent le schéma D.28 ce qui augmente l'*erreur globale* d'arrondi $r = \sum \rho_i$. Elle est alors au pire de l'ordre $r = O(\rho N)$. En prenant des pas équidistants $h = H/N$, on peut écrire $r = O(\rho \frac{H}{h})$.

L'erreur globale complète E est la somme des erreurs de la méthode plus des erreurs d'arrondi (fig. D.2)

$$E = e + r \tag{D.29}$$

Comme $e = O(h^p)$ et $r = O(\rho H/h)$, il y a un minimum pour E qui détermine une longueur de pas optimal h_{opt}.

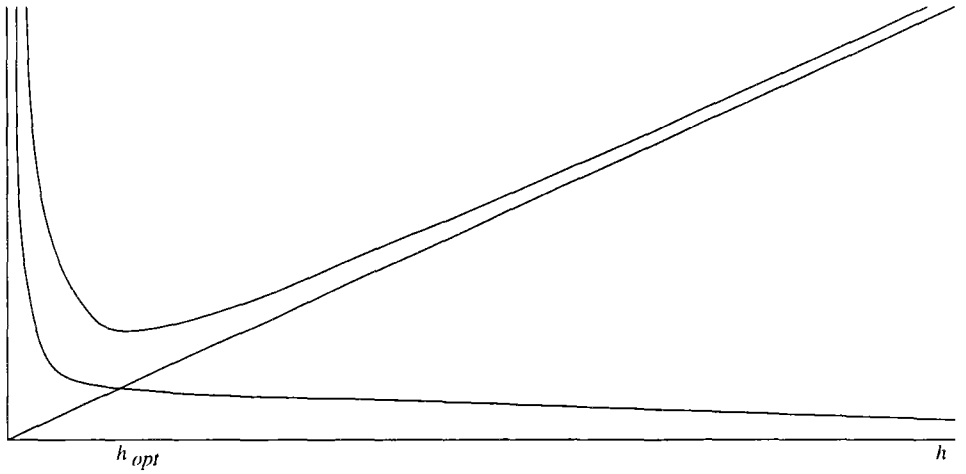


FIG. D.2 - *L'erreur d'arrondi, l'erreur de la méthode et l'erreur globale en fonction de la longueur du pas h.*

En pratique le cumul des erreurs d'arrondi est trop pessimiste. On peut supposer que les erreurs d'arrondi ne sont pas toutes dans le même sens, mais statistiquement distribués ce qui donne l'ordre $r = O(\rho\sqrt{N})$. C'est pourquoi on choisit toujours un pas $h > h_{opt}$. Remarquons aussi que pour un h trop petit ($< \rho$) la méthode d'intégration ne fonctionne plus, puisque la valeur ajoutée ne modifie plus les y_{i+1}.

D.2.3 Autres méthodes à un pas

L'estimation de l'erreur pour la méthode d'EULER-CAUCHY montre comment on peut obtenir des méthode d'un ordre supérieur à 1. Simplement on prend plus de termes dans le développement de TAYLOR (D.22). Par exemple

$$y_{i+1} = y_i + h_i f(x_i, y_i) + \frac{h_i^2}{2}[f_x(x_i, y_i) + f_y(x_i, y_i) f(x_i, y_i)] + O(h^2) \tag{D.30}$$

donne une méthode d'ordre 2. Cette méthode n'est pas pratique du tout parce qu'elle nécessite à chaque pas l'évaluation non seulement de la fonction f mais aussi les calculs de ses dérivées partielles f_x et f_y.

Méthode de HEUN et la méthode d'EULER modifiée

On peut construire des méthodes plus simples en décomposant le calcul de la méthode à un pas sur un pas fictif $\overline{y}_i$ et un pas réel y_{i+1}. Posons

$$y_{i+1} = y_i + \alpha h_i f(x_i, y_i) + \beta h_i f(\overline{x}_i, \overline{y}_i) \tag{D.31}$$

avec

$$\overline{x}_i = x_i + \gamma h_i \tag{D.32}$$
$$\overline{y}_i = y_i + \delta h_i f(x_i, y_i)) \tag{D.33}$$

où on choisit les constantes α, β, γ, et δ pour que le développement de TAYLOR possède l'ordre le plus haut possible.

$$y_{i+1} = y_i + (\alpha + \beta)h_i f(x_i, y_i) + \beta h_i^2[\gamma f_x(x_i, y_i) + \delta f_y(x_i, y_i) f(x_i, y_i)] + O(h^2) \tag{D.34}$$

La comparaison de (D.31) avec (D.34) apporte pour une méthode de deuxième ordre le système d'équations:

$$\begin{aligned} \alpha + \beta &= 1, \\ \alpha \cdot \gamma &= 1/2, \\ \beta \cdot \delta &= 1/2, \end{aligned} \tag{D.35}$$

qui a plusieurs solutions possibles.

En choisissant $\alpha = 1/2$, $\beta = 1/2$, $\gamma = 1$, $\delta = 1$, nous avons la méthode de HEUN (1900)

$$y_{i+1} = y_i + \frac{1}{2}h_i[f(x_i, y_i) + f(x_i + h_i, y_i + h_i f(x_i, y_i))]. \tag{D.36}$$

Pour le premier pas partiel, on calcule d'abord la valeur à la fin du pas virtuel $\overline{y}_i$:

$$\overline{y}_i = y_i + h_i f(x_i, y_i) \tag{D.37}$$

Puis la valeur à la fin du pas réel est obtenue en faisant la moyenne des dérivées aux début et fin du pas virtuel:

$$y_{i+1} = y_i + \frac{1}{2}h_i[f(x_i, y_i) + f(x_i + h_i, \overline{y}_i)] \tag{D.38}$$

Le pas (virtuel) peut être considéré comme un premier essai qui est corrigé avec la formule D.38. (voir méthodes *prédicteur-correcteur.*)

La méthode d'EULER *modifiée* (qui est quelques fois appellée la méthode du *point milieu*) (1960 [25]) est obtenue en choisissant $\alpha = 0$, $\beta = 1$, $\gamma = 1/2$, et $\delta = 1/2$,

$$y_{i+1} = y_i + h_i f(x_i + \frac{h_i}{2}, y_i + \frac{h_i}{2} f(x_i, y_i)). \tag{D.39}$$

En utilisant la dérivée au point initial, on calcule un point au milieu de l'intervalle à traverser

$$\overline{y}_i = y_i + \frac{h_i}{2} f(x_i, y_i). \tag{D.40}$$

Avec la dérivée à ce point, on calcule le pas entier (cf. fig. D.3).

$$y_{i+1} = y_i + h_i f(x_i + \frac{h_i}{2}, \overline{y}_i) \tag{D.41}$$

Les deux méthodes nécessitent deux évaluations de la fonction f et les erreurs des méthodes sont de l'ordre $O(h^2)$.

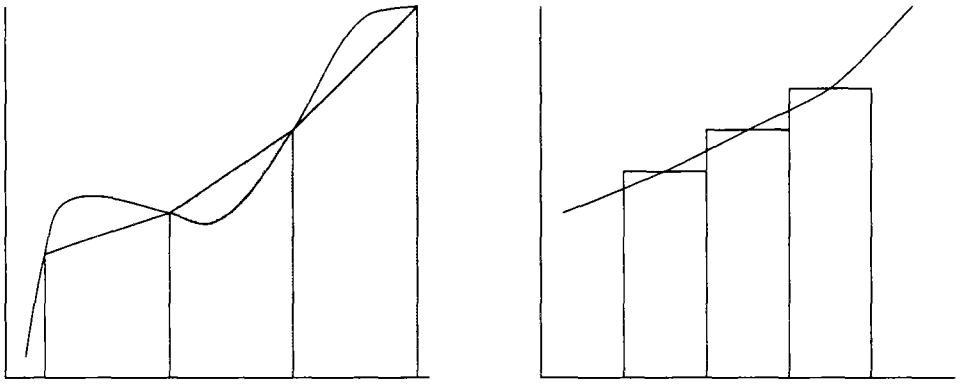


FIG. D.3 - *La méthode de* HEUN *et la méthode d'*EULER *modifiée.*

Méthode de RUNGE-KUTTA

On peut généraliser l'approche (D.31) pour prendre en compte p termes du développement de TAYLOR. Nous commençons avec l'équation

$$\begin{aligned} y_{i+1} &= y_i + \sum_{j=1}^{p} \alpha_j k_j \quad \text{avec} \\ k_j &= h_i f(x_i + \beta_j, y_i + \sum_{l=1}^{p} \gamma_{jl} k_l) \\ y^*_{i+1} &= y_{i+1} + O(h^p). \end{aligned} \tag{D.42}$$

Nous comparons cette approche D.43 avec un développement de TAYLOR jusqu'à l'ordre p.

Exemple: Si on prend $p = 1$, $\alpha = 1$, $\beta = 0$ et $\gamma_{11} = 0$, on retrouve la méthode d'EULER D.20

$$\begin{aligned} k_1 &= h_i f(x_i + 0 \cdot h_i, y_i + 0) \\ y_{i+1} &= y_i + 1 \cdot k_1. \end{aligned} \tag{D.43}$$

C'est pourquoi on appelle parfois dans la littérature toutes les méthodes à un pas des méthodes de RUNGE-KUTTA.

La méthode classique de RUNGE-KUTTA (1895/1901) est d'ordre $p = 4$. La comparaison avec le développement de TAYLOR mène à un système d'équations qui contient deux paramètres de plus que le nombre d'équations disponibles pour les déterminer. Les deux paramètres libres sont choisis de façon à ce que la formule devienne symétrique (cf. [32, chapt. 10.3.4]). Le schéma de RUNGE-KUTTA est alors défini par

$$\begin{aligned} k_1 &= h_i f(x_i, y_i) \\ k_2 &= h_i f(x_i + \frac{h_i}{2}, y_i + \frac{k_1}{2}) \\ k_3 &= h_i f(x_i + \frac{h_i}{2}, y_i + \frac{k_2}{2}) \end{aligned} \tag{D.44}$$

$$\begin{aligned} k_4 &= h_i f(x_i + h_i, y_i + k_3) \\ y_{i+1} &= y_i + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4 \\ y^*_{i+1} &= y_{i+1} + O(h^4). \end{aligned}$$

Ce schéma fait quatre évaluations de la fonction f pour déterminer une nouvelle approximation y_{i+1} par une moyenne pondérée. Comme l'erreur de la méthode est d'ordre $O(h^4)$ l'erreur diminue vite si on diminue le pas h. Mais elle augmente aussi vite, si on augmente le pas. Le choix d'une bonne longueur de pas a une influence importante sur la précision de cette méthode. (voir plus loin)

Une estimation de l'erreur est très compliquée. Elle est donnée avec des démonstrations de convergence en [44, p. 73-80 et p. 87-91] et en [25, p. 71/72].

Remarque: On pourrait imaginer des méthodes d'ordre supérieur à 4. Elles existent, mais elles ont l'inconvenient de nécessiter toujours au moins $p + 1$ évaluations de la fonction f pour un ordre $O(h^p)$.

D.2.4 Comparaison des méthodes à un pas

Pour comparer [14, p. 113/114] les trois schémas: *le schéma d'*EULER D.20, *le schéma d'*EULER *modifié* D.39, et *le schéma de* RUNGE-KUTTA D.44 pour la solution d'un problème à condition initiale $df/dy = f(x, y)$, $y(0) = 0$ dans l'intervalle $0 \leq x \leq 1$, nous supposons les erreurs des schémas vérifiant $3h_{EC}$, $11h^2_{Emod}$, et $42h^4_{RK}$. Si nous revendiquons une erreur $\varepsilon < 10^{-8}$, les longueurs de pas doivent vérifier $3h_{EC} < 10^{-8}$, $11h^2_{Emod} < 10^{-8}$, et $42h^4_{RK} < 10^{-8}$. Ou si on compare les nombres des pas nécessaires pour chacune des méthodes N_{EC}, N_{Emod}, N_{RK} pour intégrer la fonction sur l'intervalle donnée, il faut que les relations vérifient

$$\begin{aligned} N_{EC} &> 3 \cdot 10^8 &&= 300.000.000, \\ N_{Emod} &> \sqrt{11} \cdot 10^4 &&\approx 33.166, \\ N_{RK} &> \sqrt[4]{42} \cdot 10^2 &&\approx 253. \end{aligned} \tag{D.45}$$

C'est un exemple qui montre bien la supériorité de la performance de la méthode de RUNGE-KUTTA [14]. Il faut remarquer que l'on fait dans cette méthode plusieurs évaluations de la fonction f à chaque pas. Néanmoins, la méthode de RUNGE-KUTTA, malgré ses quatre évaluations, nécessite l'effort de calcul le plus faible pour parcourir l'intervalle.

Remarque: si f ne dépend pas de y (ce qui revient à l'intégration numérique d'une fonction normale), la méthode de HEUN correspond à l'intégration par trapèzes, la méthode d'EULER *modifiée* correspond à la règle du *point de milieu* et la méthode de RUNGE-KUTTA correspond à la règle de SIMPSON.

D.2.5 Contrôle de la précision

Si on peut déterminer l'erreur, on peut observer la précision des calculs au cours d'une intégration. Contrôler la précision veut dire que l'on cherche

à conserver l'erreur inférieure à une certaine limite. Comme l'erreur dépend du pas et de l'ordre d'une méthode, nous avons deux moyens pour piloter une intégration.

Pour minimiser l'effort de calcul, il faut évidement minimiser le nombre de pas. Les pas sont alors plus longs. Par ailleur on obtient des erreurs petites par des petits pas. La connaissance de l'erreur locale n'a pas de valeur pratique pour l'estimation de l'erreur réel de la discrétisation ou d'arrondi. C'est pourquoi on utilise le principe suivant. (Le changement de la longueur du pas h_i en fonction de la précision réalisée au pas réel précédent.)

Comme on a vu dans les sections précédentes la valeur d'une approximation $y_i(x)$ dépend toujours de la longueur des pas avec lesquels y a été calculé. Nous allons le spécifier alors par $y_i(x_i; h)$. Après avoir trouvé une approximation $y_i(x_i; h)$ avec un pas h nous allons calculer pour le même x_i, mais avec un autre pas (disons $h/2$) l'approximation $y_i(x_i; h/2)$. En utilisant la première approximation de l'erreur globale $e(x) = e_p h^p + e_{p+1} h^{p+1} + \cdots$ (équation D.27 et [75, p. 419]) nous pouvons maintenant écrire

$$y_i(x_i; h) - y^*(x_i) = e_p(x) \cdot h^p \tag{D.46}$$

$$y_i(x_i; \frac{h}{2}) - y^*(x_i) = e_p(x) \cdot \left(\frac{h}{2}\right)^p \tag{D.47}$$

La différence entre les deux équations D.46 - D.47 est

$$y_i(x_i; h) - y_i(x_i; \frac{h}{2}) = e_p(x) \cdot \left(\frac{h}{2}\right)^p (2^p - 1) \tag{D.48}$$

que l'on résoud en e_p

$$e_p(x) \left(\frac{h}{2}\right)^p = \frac{y_i\,(x_i; h) - y_i(x_i; \frac{h}{2})}{2^p - 1}, \tag{D.49}$$

avec la réintégration de l'équation D.47

$$y_i(x_i; \frac{h}{2}) - y^*(x_i) = \frac{y_i(x_i; h) - y_i(x_i; \frac{h}{2})}{2^p - 1}. \tag{D.50}$$

L'équation D.50 est une bonne estimation de l'erreur globale de la méthode. Maintenant nous connaissons l'expression de l'erreur, il faut maintenir au-dessous d'une certaine limite $\mathcal{E}$. En faisant un pas de longueur h_1 nous commettons une erreur $\Delta_1 = |\frac{y(x;h_1) - y(x;h_1/2)}{2^p - 1}|$ D.50. Pour obtenir une erreur Δ_2, il faudrait faire un pas h_2 qui peut être déterminé par la formule

$$\frac{h_2}{h_1} = \sqrt[p+1]{\frac{2^p - 1}{2^p} \frac{\Delta_2}{\Delta_1}}. \tag{D.51}$$

Elle est établie en prenant en compte l'erreur locale d'ordre $(O(h^{p+1}))$ ([75] p. 423/424, [66] p. 574/575).

Si on choisit pour Δ_2 une constante $\mathcal{E}$ qui représente l'erreur permise maximale, on peut déterminer si le pas choisi respecte la limite. Sinon, l'équation

D.51 indique avec quelle longueur h_2 il faut répéter le pas précédent. Si Δ_1 était plus petit que $\mathcal{E}$, le pas était bon et on peut utiliser h_2 pour le prochain pas en supposant que la fonction n'a pas de courbure forte. C'est pourquoi on utilise en pratique pour la méthode de RUNGE-KUTTA ($O_{RK}(h^{p=4})$) une autre formule qui est

$$h_{nouveau} = \begin{cases} Sh_1 \sqrt[5]{\frac{\mathcal{E}}{\Delta_1}} & \mathcal{E} > \Delta_1 \\ Sh_1 \sqrt[4]{\frac{\mathcal{E}}{\Delta_1}} & \mathcal{E} < \Delta_1 \end{cases} . \tag{D.52}$$

Parce qu'on ne connaît pas bien l'erreur réelle (Δ_1 est une approximation), on a ajouté un facteur de sécurité S qui vaut à peu près $0,9 \ldots 0,98 (\approx \sqrt[5]{15/16})$.

Il y a des façons différentes de choisir la constante $\mathcal{E}$ pour la comparaison de l'erreur. Si ce choix est fait en fonction de h même (voir plus loin), prendre la racine 5^e n'est plus correct. Quand on diminue maintenant le pas h, la nouvelle valeur $h_{nouveau}$ va échouer aussi, parce que la constante $\mathcal{E}$ de comparaison a changé dans les mêmes proportions. Prendre la racine 4^e (qui est encore assez proche de la 5^e) mène en pratique à des résultats satisfaisants ([66] chap. 15.2).

Comme déjà mentionné, le choix de la constante $\mathcal{E}$ dépend du problème. Si on traite un ensemble d'équations dont les variables dépendantes ont des grandeurs très différentes, on va choisir une constante relative aux valeurs obtenues $\mathcal{E} = \epsilon \cdot y$, où ϵ est 10^{-8} par exemple. Par contre, si on a des équations qui passent par zéro, mais qui ont une valeur maximale, il vaut mieux fixer $\mathcal{E}$ a une valeur $\epsilon \cdot \max |y|$.

Les critères précédents pour le choix de $\mathcal{E}$ sont locaux. Si on veut surtout un contrôle de l'erreur globale, il faut imposer un $\mathcal{E}$ qui diminue avec le pas h, parce qu'il y aura davantage de pas entre le point de départ x_0 et le point cible x_N. Dans de tels cas on prend souvent $\mathcal{E} = \epsilon \cdot h \cdot f$. Cette prescription donne une comparaison relative aux *incréments* de la fonction y.

Remarque: Il y a aussi des schémas de contrôle du pas qui ne font pas la comparaison de deux pas distincts de la même méthode mais plutôt une comparaison entre deux méthodes d'un ordre différent $O(h^p)$ et $O(h^{p-1})$ sur le même pas. Comme les valeurs intermédiaires (par exemple $f(x_{i+1}, y_i)$) sont déjà calculées, on peut les utiliser pour une autre méthode d'un ordre inférieur sans coût de calcul supplémentaire. Ces méthodes sont désignés sous le nom de méthodes de RUNGE-KUTTA-FEHLBERG [75, p. 426-429], [32, chap. 10.3.4.3] ou méthodes emboitées [26, p. 108-110].

D.2.6 Extrapolation de RICHARDSON et la méthode de BULIRSCH-STOER-GRAGG

Généralement, on augmente la précision des calculs en diminuant le pas h. Si on extrapole le résultat calculé dans la direction de pas beaucoup plus petits, qu'ils n'ont réellement été, on améliore la précision du résultat. Le but est l'extrapolation vers un pas $h = 0$.

L'idée est de considérer le résultat d'un calcul numérique des $y(x)$ comme une fonction analytique d'un paramètre variable (le pas h). On évalue cette fonction avec quelques h différents, dont aucun ne satisfait les bornes de précision souhaitée. Dés que nous connaissons suffisamment la fonction, nous la remplaçons par une forme analytique, dont nous prenons la valeur au point $h = 0$. (fig. D.4)

La deuxième idée concerne le type de fonction analytique utilisé pour faire l'extrapolation. C'est une extrapolation par une fonction rationnelle

$$f(x) = \frac{a_p x^p + a_{p-1} x^{p-1} + \cdots + a_0}{b_l x^l + \cdots + b_0}$$

et l'astuce de GRAGG (« GRAGG's trick» [75, p. 454]) (voir éq. D.55).

Et finalement, le troisième point est l'utilisation d'une méthode dont l'erreur est toujours d'un ordre pair. Cela permet une approximation par fonction rationnelle en termes de h^2 au lieu de h.

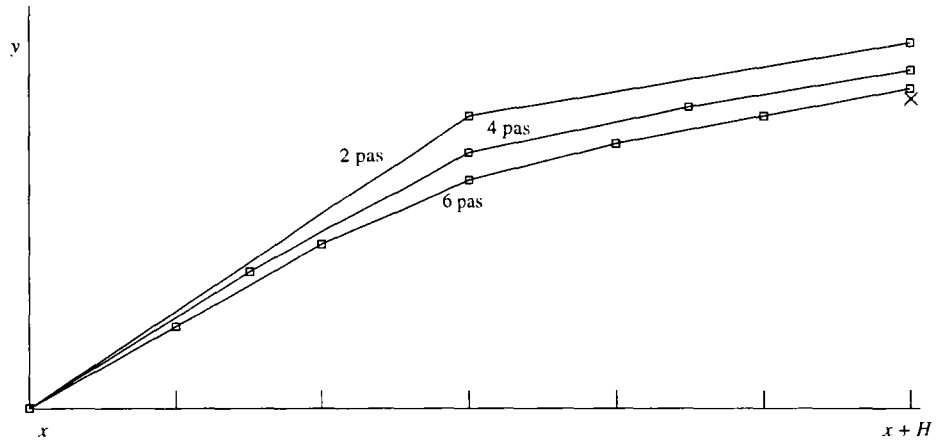


FIG. D.4 - *L'extrapolation de* RICHARDSON

La méthode de BULIRSCH-STOER-GRAGG consiste à rassembler les trois idées précédentes dans un seul schéma. Cela peut rester une bonne approximation des fonctions, même si on a déjà dépassé des valeurs comparables pour le pas h. Cela veut dire que h peut devenir si grand que la notion "ordre" perd son sens et la méthode marche toujours bien. Un pas de BSG noté h_{BSG} peut être très long, mais il est composé d'un grand nombre de pas partiels δ_l. On remarquera que le pas de la méthode est toujours noté h et que les pas intermédiairs sont notés δ.

Pour un problème du type $y' = f(x, y)$, $y_0 = y(x_0)$, on fait une suite d'essais $j = 1, 2, \ldots$ de parcours de l'intervalle $[x_i, x_i + h]$ avec des nombre M_j différents (M pair) de pas partiels $l = 1, \ldots, M_j$. Soit $\delta = h/M$ et $x_l = x_i + l\delta$ on calcule les grandeurs intermédiaires:

$$\begin{aligned} z_0 &= y_i \\ z_1 &= y_i + \delta f(x_i, y_i), \\ z_{l+1} &= z_{l-1} + 2\delta f(x_l, z_l), \quad l = 1, 2, \ldots M \end{aligned} \tag{D.53}$$

Si on compare cette équation avec les schémas D.41 et D.70, on retrouve une modification de la méthode du point de milieu pour le calcul des points z ($\delta_{(D.53)} = 2h_{(D.41)}$).

Après chaque essai pour un M_j, on fait une extrapolation par fonction rationnelle. Cette extrapolation nous fournit une valeur extrapolée (bien sûr) et une estimation de l'erreur. Si l'erreur n'est pas assez petite, on continue la suite avec un M_{j+1} supérieur. Si l'erreur est satisfaisante, le pas BSG est fini et on peut entamer le prochain pas $i+1$.

En pratique on utilise pour les M_j la suite suivante

$$M_j = 2, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, \ldots \; (M_j = 2M_{j-2}) \tag{D.54}$$

L'extrapolation des résultats vers un pas $\delta \to 0$ est effectuée avec la fonction[2]

$$G(x_{i+1}, \delta) = \frac{1}{2}(z_M + z_{M-1} + \delta f(x_{i+1}, z_M)) \tag{D.55}$$

qui est utilisée comme

$$T_j^0 = G(x_{i+1}, \delta_j) \tag{D.56}$$

où l'indice supérieur indique la colonne dans le schéma suivant D.58. On calcule

$$T_j^k = \frac{1}{2^{2k}-1}(2^{2k}T_j^{k-1} - T_{j-1}^{k-1}), \quad 1 \le k \le j. \tag{D.57}$$

dans le schéma

$$\begin{array}{lllll}
G(x_{i+1}, \delta_0) = & T_0^0 & & & \\
 & & T_1^1 & & \\
G(x_{i+1}, \delta_1) = & T_1^0 & & T_2^2 & \\
 & & T_2^1 & & T_3^3 \\
G(x_{i+1}, \delta_2) = & T_2^0 & & T_3^2 & \\
 & & T_3^1 & & \\
G(x_{i+1}, \delta_3) = & T_3^0 & & &
\end{array} \tag{D.58}$$

On peut démontrer que les colonnes du schéma D.58 convergent [75, chap. 3.5, chap. 7.2.14]

$$\lim_{j \to \infty} T_j^k = y(x_{i+1}) \tag{D.59}$$

On détermine l'erreur du pas j par la différence entre les résultats de deux essais T_j^j et T_{j-1}^{j-1}. L'estimation de l'erreur peut être utilisée pour l'arrêt de la suite des j (l'application du schéma D.58) mais aussi pour le contrôle du pas global h (l'application d'un pas BSG). Des information complémentaires pour le contrôle du pas global sont données dans [66, chap. 15.4].

L'avantage de la méthode de BULIRSCH-STOER-GRAGG est sa grande précision pour une évaluation minimale des fonctions différentielles. Elle est stable, même très près des pôles.

2. On appelle cette fonction la *fonction de* GRAGG. Elle est analogue à la formule d'EULER-MCLAURIN pour le calcul numérique des intégrales. De la même façon, le schéma D.58 est analogue à la méthode de ROMBERG.

D.2.7 Méthodes multi-pas

Par rapport aux méthodes à un pas qui calculent un nouveau y_{i+1} en fonction d'une unique valeur précédente, les méthodes multi-pas stockent les solutions à chaque pas et elles utilisent ces informations pour une extrapolation du prochain pas. Cette extrapolation est appellée le pas partiel *prédicteur*. Ensuite le résultat est corrigé avec un pas d'une méthode *implicite* qui utilise déjà l'information sur la dérivée au nouveau point. Ce deuxième pas partiel est designé *correcteur*. Des itérations (répétition du deuxieme pas) sont quelques fois nécessaires, si l'on n'a pas encore atteint la limite de l'erreur permise. Dans tous les cas, les méthodes multi-pas ne nécessitent qu'une seule évaluation des fonctions différentielles par pas. Leur stabilité est faible, surtout proche des discontinuités.

Méthodes explicites

On calcule une approximation de la valeur $y_{j+r} = y(x_{j+r})$ en fonction de $r \geq 2$ valeurs données $y_l, l = j, j+1, \ldots, j+r-1$ aux points équidistants $x_l = x_0 + lh$

$$y_j, y_{j+1}, \cdots, y_{j+r-1} \Rightarrow y_{j+r} \tag{D.60}$$

Pour l'initialisation des méthodes multi-pas il faut alors connaitre $r = j + k$ points, qu'il faut avoir calculés par une autre méthode.

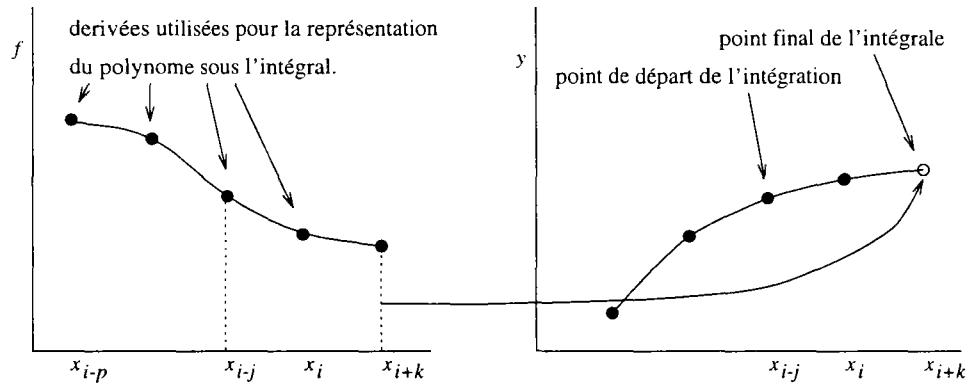


FIG. D.5 - *La notation des points pour les méthodes multi-pas.*

Par l'intégration de $y' = f(x, y)$ on obtient

$$y^*(x_{i+k}) - y^*(x_{i-j}) = \int_{x_{i-j}}^{x_{i-k}} f(x, y(x))\, dx, \tag{D.61}$$

Dans cette équation l'index k indique la distance entre le point actuel x_i et le point à calculer x_{i+k}. $k = 1$ est le cas typique pour calculer le prochain point (pas prédicteur), $k = 0$ designe une méthode implicite (pas correcteur). (Fig. D.5)

Maintenant on remplace la fonction à intégrer par un polynôme d'interpolation $P_p(x)$ avec

$$\begin{aligned} \deg P_p(x) &\leq p, \\ P_p(x_l) &= f(x_l, y(x_l)), \quad l = i, i-1, \ldots, i-p. \end{aligned}$$

et on écrit

$$y(x_{i+k}) - y(x_{i-j}) = \int_{x_{i-j}}^{x_{i+k}} P_p(x)\,dx. \tag{D.62}$$

Pour l'intégration du polynôme on utilise la formule de LAGRANGE de l'interpolation ([75] chap. 2.1.1, [26] p. 5-7)

$$\begin{aligned} y(x_{i+k}) - y(x_{i-j}) &\approx \sum_{l=0}^{p} f(x_{i-l}, y_{i-l}) \int_{x_{i-j}}^{x_{i+k}} L_l(x)\,dx && \text{(D.63)} \\ &= h \sum_{l=0}^{p} b_{p,l} f(x_{i-l}, y_{i-l}) && \text{(D.64)} \end{aligned}$$

avec

$$\begin{aligned} b_{p,l} &= \frac{1}{h} \int_{x_{i-j}}^{x_{i+k}} L_l(x)\,dx && \text{(D.65)} \\ &= \int_{-j}^{k} \prod_{n=0;n\neq l}^{p} \frac{s+n}{-l+n}\,ds; \quad l = 0, 1, \ldots, p. \end{aligned}$$

Le schéma général pour les méthodes multi-pas s'écrit alors:

$$y_{i+k} = y_{i-j} + h \sum_{l=0}^{p} b_{p,l} f(x_{i-l}, y_{i-l}). \tag{D.66}$$

Les differents choix pour les constantes k, j, p mènent aux differentes méthodes multi-pas.

Pour $k = 1$, $j = 0$, et $p = 0, 1, \ldots$ on dispose des méthodes de ADAMS-BASHFORTH

$$\begin{aligned} y_{i+1} &= y_i + h[b_{p,0} f_i + b_{p,1} f_{i-1} + \cdots + b_{p,p} f_{i-p}] && \text{(D.67)} \\ b_{p,l} &= \int_0^1 \prod_{n=0;n\neq l}^{p} \frac{s+n}{-l+n}\,ds; \quad l = 0, 1, \ldots, p. \end{aligned}$$

Pour ces méthodes, il faut connaître $r = p + 1$ points précédents. On trouve dans le tableau D.6 quelques constantes b pour les premiers ordres. On peut remarquer que le cas $p = 0$ équivaut la méthode d'EULER.

Méthodes implicites

Si on choisit $k = 0$, $j = 1$, et $p = 0, 1, \ldots$ et si on remplace i par $i + 1$, on trouve les méthodes d'ADAMS-MOULTON

$$\begin{aligned} y_{i+1} &= y_i + h[b_{p,0} f_{i+1} + b_{p,1} f_i + \cdots + b_{p,p} f_{i+1-p}] && \text{(D.68)} \\ b_{p,l} &= \int_0^1 \prod_{n=0;n\neq l}^{p} \frac{s+n}{-l+n}\,ds; \quad l = 0, 1, \ldots, p. \end{aligned}$$

$b_{p,l}$	0	1	2	3	4	5
$b_{0,l}$	1					
$b_{1,l}$	3/2	-1/2				
$b_{2,l}$	23/12	-16/12	5/12			
$b_{3,l}$	55/24	-59/24	37/24	-9/24		
$b_{4,l}$	1901/720	-2774/720	2616/720	-1274/720	251/720	
$b_{5,l}$	4277/1440	-7923/1440	9982/1440	-7298/1440	2877/1440	-475/1440

FIG. D.6 - *Les constantes $b_{p,l}$ du schéma d'*ADAMS-BASHFORTH *jusqu'à l'ordre 5.*

$b_{p,l}$	0	1	2	3	4	5
$b_{0,l}$	1					
$b_{1,l}$	1/2	1/2				
$b_{2,l}$	5/12	8/12	-1/12			
$b_{3,l}$	9/24	19/24	-5/24	1/24		
$b_{4,l}$	251/720	646/720	-264/720	106/720	-19/720	
$b_{5,l}$	475/1440	1427/1440	-798/1440	482/1440	-173/1440	27/1440

FIG. D.7 - *Les constantes $b_{p,l}$ du schéma d'*ADAMS-MOULTON *jusqu'à l'ordre 5.*

y_{i+1} et f_{i+1} apparaissent maintenant des deux cotés de l'équation D.68. Sa résolution nécessite une procédure itérative. On dit que les méthodes d'ADAMS-MOULTON sont *implicites* par rapport aux méthodes d'ADAMS-BASHFORTH qui sont *explicites*. Quelques valeurs pour les coefficients b d'ADAMS-MOULTON se trouvent dans tableau D.7.

Pour déterminer un y_{i+1} on utilise une méthode *explicite* de type ADAMS-BASHFORTH comme *predicteur* $y_{i+1}^{(0)}$ et puis on fait l'itération[3] avec une méthode implicite de type ADAMS-MOULTON pour calculer les $y_{i+1}^{(n)}$. La dénommation *correcteur* est utilisée car souvent un seul pas d'itération est suffisant. (On prend $y_{i+1} = y_{i+1}^{(1)}$.)

Dans la méthode de NYSTRÖM on choisit $k = 1$, $j = 1$ et on obtient

$$\begin{aligned} y_{i+1} &= y_{i-1} + h[b_{p,0} f_i + b_{p,1} f_{i-1} + \cdots + b_{p,p} f_{i-p}] \qquad \text{(D.69)} \\ b_{p,l} &= \int_{-1}^{1} \prod_{n=0;n\neq l}^{p} \frac{s+n}{-l+n}\, ds; \quad l = 0, 1, \ldots, p. \end{aligned}$$

C'est une méthode *prédicteur* avec $r = p + 1$. Ici il faut noter que si on prend $p = 0$ le schéma D.69 devient la *règle du point de milieu.*

$$y_{i+1} = y_{i-1} + 2hf_i \qquad \text{(D.70)}$$

3. L'indice supérieure $^{(n)}$ indique le pas d'itération

Les méthodes de MILNE sont des schémas correcteurs. On les obtient en prenant $k = 0$ et $j = 2$, si on remplace i par $i + 1$.

$$\begin{aligned} y_{i+1} &= y_{i-1} + h[b_{p,0} f_{i+1} + b_{p,1} f_i + \cdots + b_{p,p} f_{i+1-p}] \\ b_{p,l} &= \int_{-2}^{1} \prod_{n=0;n\neq l}^{p} \frac{s+n}{-l+n}\, ds; \quad l = 0, 1, \ldots, p. \end{aligned} \tag{D.71}$$

Elles sont également résolues par itération.

Parce qu'on ne connaît pas *a priori* le comportement des solutions des ED, on est obligé d'utiliser un pas variable de point à point pour borner l'erreur dans sa grandeur maximale. Les méthodes multi-pas qui utilisent des points équidistants et un ordre fixe ne sont pas très utiles en pratique. Le changement du pas nécessite de recalculer des données aux points précédents pour l'extrapolation ce qui réduit fortement l'utilité de ces méthodes.

En pratique on utilise la formule de NEWTON pour l'interpolation dans D.62 ce qui a quelques avantages pour le contrôle du pas [75, chap. 2.1.3.4]. On peut construire des schémas prédicteur/correcteur comme avec la formule de LAGRANGE (cf. équation D.62+2), mais le pas d'itération n'est effectué qu'une seule fois. Puis on compare la valeur du prédicteur $f_{i+1}^{(0)}$ avec la valeur du correcteur $f_{i+1}^{(1)}$. La différence est utilisée pour calculer une estimation de l'erreur E_p. On peut juger si le pas précédent a été assez précis avec la même formule pour le contrôle du pas que celle du § D.2.5.

$$h_{\text{nouveau}} = h_{\text{ancien}} \sqrt[p-2]{\frac{\mathcal{E}}{|E_p|}} \tag{D.72}$$

On peut combiner cette stratégie avec un changement de l'ordre p. On calcule les trois valeurs

$$\left(\frac{\mathcal{E}}{|E_{p-1}|}\right)^{1/(p+1)}, \left(\frac{\mathcal{E}}{|E_p|}\right)^{1/(p+2)}, \left(\frac{\mathcal{E}}{|E_{p+1}|}\right)^{1/(p+3)}, \tag{D.73}$$

et on cherche la valeur maximale. Si c'est la première, on diminue p de 1. Si la deuxième est le maximum on continue avec le même p, et si la troisième est la plus grande, on augmente p de 1. Le point remarquable est que l'on peut calculer les E_{p-1}, E_p, et E_{p+1} récursivement à partir des valeurs existantes ([75] chap. 7.2.13).

Cette méthode de contrôle est denommée "self-starting", puisqu'on commence avec un ordre 0, et le prochain pas est augmente à 1, ...

D.2.8 Systèmes raides (*stiff problems*)

Il y a une classe de SED qui – bien que théoriquement solubles – entraînent des problèmes numériques graves. C'est le cas, quand il y a au moins deux

grandeurs très différentes de la variable indépendante. Les valeurs propres du système sont très éloignées. Si on prend l'exemple

$$\begin{array}{rcl} u' &=& 998u + 1998v \\ v' &=& -999u - 1999v \end{array} \quad \text{avec les conditions initiales} \quad \begin{array}{rcl} u(0) &=& 1 \\ v(0) &=& 0 \end{array} \tag{D.74}$$

nous trouvons la solution analytique

$$\begin{array}{rcl} u^* &=& 2e^{-x} - e^{-1000x} \\ v^* &=& -e^{-x} + e^{-1000x}. \end{array} \tag{D.75}$$

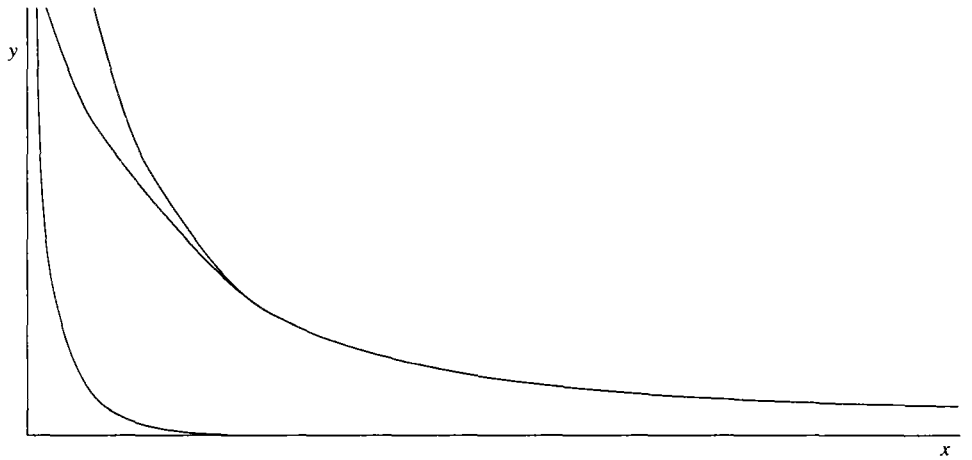


FIG. D.8 - *La solution globale est composée de deux courbes, dont une est presque zéro.*

La solution obtenue par la méthode d'EULER peut être écrite

$$\begin{array}{rcl} u_i &=& 2(1-h)^i - (1-1000h)^i \\ v_i &=& -1(1-h)^i + (1-1000h)^i \end{array} \tag{D.76}$$

Il est évident que les approximations convergent seulement, si le pas h est assez petit pour que

$$|1-h| < 1 \quad \text{et} \quad |1-1000h| < 1. \tag{D.77}$$

L'influence du deuxième terme sur la solution globale devient negligeable quand on a dépassé une certaine limite (voir fig. D.8). Malheureusement ce n'est pas vrai pour l'intégration numérique. Bien que e^{-1000x} n'apporte plus rien à la solution, il nous faut un $h < 1/1000$ pour que la méthode reste stable. Ce comportement numérique est appelé un « problème de raideur» (stiff problem). Toutes les méthodes présentées jusqu'ici échouent face à de tels problèmes qui sont nombreux dans les domaines des réseaux électriques, de la cinétique chimique et ceux des transports de chaleur et masse dans les processus industriels.

GEAR a développé des critères pour la construction des méthodes qui sont stables aussi pour l'intégration des SEDs raides ([41], [32, chapt. 11.4.3 / 10.6]). Ce sont des méthodes de différenciation rétrograde.

Pour un système linéaire avec une matrice définie positive $\boldsymbol{A}$

$$\boldsymbol{y}' = -\boldsymbol{A}\boldsymbol{y} \quad \boldsymbol{y}(0) = \boldsymbol{y}_0 \tag{D.78}$$

on trouve par différenciation explicite la méthode d'EULER-CAUCHY D.20

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h\boldsymbol{y}_i' \tag{D.79}$$

ou

$$\boldsymbol{y}_{i+1} = (\boldsymbol{I} - \boldsymbol{A}h)\boldsymbol{y}_i \tag{D.80}$$

Une matrice $\boldsymbol{C}^i$ tend vers 0 pour $i \to \infty$, si la plus grande valeur propre de $\boldsymbol{C}$ est inférieure à 1. Cela veut dire, que la méthode d'EULER-CAUCHY converge, si la plus grande valeur propre en valeur absolue de $(\boldsymbol{I} - \boldsymbol{A}h)$ est inférieure à 1 ou, en notant λ les valeurs propres de $\boldsymbol{A}$, si h est inférieur à $\frac{2}{\lambda_{max}}$. A l'opposé, une différenciation rétrograde (ou différenciation implicite) de D.78 mène à

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h\boldsymbol{y}_{i+1}' \tag{D.81}$$

ou

$$\boldsymbol{y}_{i+1} = (\boldsymbol{I} + \boldsymbol{A}h)^{-1}\boldsymbol{y}_i \tag{D.82}$$

Les valeurs propres de $(\boldsymbol{I} + \boldsymbol{A}h)^{-1}$ sont alors $(1 + \lambda h)^{-1}$ qui sont toujours inférieures à 1 quelque soit h ($h > 0$, $\lambda \geq 0$).

En utilisant ce schéma on obtient la stabilité de l'intégration. Mais on paye cher en efforts de calculs (inversion d'une matrice à chaque pas).

Pour le problème général $\boldsymbol{y}' = \boldsymbol{f}(x, \boldsymbol{y})$ la différenciation implicite mène à

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h\boldsymbol{f}(x_{i+1}, \boldsymbol{y}_{i+1}). \tag{D.83}$$

C'est un ensemble d'équations non linéaires qu'il faut résoudre par itération à chaque pas. Pour faciliter le travail, on peut linéariser cette formule et on obtient:

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h\left[\boldsymbol{f}(x_{i+1}, \boldsymbol{y}_i) + \left.\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{y}}\right|_{\boldsymbol{y}_i} \cdot (\boldsymbol{y}_i - \boldsymbol{y}_{i+1})\right] \tag{D.84}$$

ou

$$\boldsymbol{y}_{i+1} = \left[\boldsymbol{y}_i + h\left(\boldsymbol{f}(x_{i+1}, \boldsymbol{y}_i) - \left.\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{y}}\right|_{\boldsymbol{y}_i} \cdot \boldsymbol{y}_i\right)\right]\left(\boldsymbol{I} - h\left.\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{y}}\right|_{\boldsymbol{y}_i} \boldsymbol{y}_i\right)^{-1}. \tag{D.85}$$

qui est appelée une formule semi-implicite. $\partial \boldsymbol{f}/\partial \boldsymbol{y}|_{\boldsymbol{y}_i}$ est la matrice des dérivées partielles appelée matrice *Jacobienne* qu'il faut calculer à chaque pas avec l'inverse de $\boldsymbol{I} - h\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{y}}$. Théoriquement l'équation D.84 n'est pas stable, mais en pratique elle l'est dans la plupart de cas, parce qu'elle est localement similaire

à l'équation D.82 (La matrices des constantes A est remplacée par la matrice Jacobienne).

Les méthodes présentées ici pour les problèmes raides sont du premier et deuxième ordre. Mais on a développé aussi des méthodes d'un ordre supérieure (ou même d'un ordre variable) qui ont repris les idées des méthodes de RUNGE-KUTTA ou des méthodes multipas (appelées les méthodes de GEAR (1971). Outre les méthodes de GEAR, la méthode de KAPS-RENTROP (1979) est très performante grâce à sa stabilité et au contrôle du pas ([75] p. 464/465).

D.2.9 Conclusion

A l'université de Toronto on a été développé un programme DETEST [47] en vue de comparer les différentes méthodes. Des nombreuses expérimentations ont été réalisées, dont les résultats sont publiés dans [33]. L'utilité d'une méthode est déterminée par les efforts de calcul qu'elle demande. Ils sont classés en

- effort pour calculer les fonctions f qui dépend de leur nombre m et de leurs structures,
- effort pour le contrôle de la précision (changement de pas et d'ordre),
- les autres calculs.

L'effort minimal pour l'évaluation des fonctions f est obtenu avec les méthodes *prédicteur-correcteur*. Le pas *prédicteur* ne calcule qu'une seule fois les f et le pas correcteur les évalue autant de fois qu'il fait d'itérations (habituellement une ou deux fois). Les efforts pour le contrôle du pas et les autres actions annulent malheureusement cet avantage. C'est pourquoi elles sont utilisées de moins et moins dans les applications normales. Si les f sont vraiment très compliquées ou si on souhaite une précision extrême ($10^{-20} < \varepsilon < 10^{-30}$), elles retrouvent leur utilité. Des développements récents relatifs aux schémas à ordre variable bien qu'encore plus compliqués apportent une efficacité supplémentaire.

Les méthodes d'extrapolation nécessitent le moins de calcul si l'on exclut des évaluations des f. La difficulté est de trouver un bon algorithme pour le contrôle du pas global. Tous les algorithmes connus marchent bien, mais quelque fois le changement de pas n'est pas assez sensible et les calculs sont effectués avec trop de précision.

Pour les problèmes qui ne nécessitent que peu d'précision ($\varepsilon \approx 10^{-4}$) la méthode de RUNGE-KUTTA est préférable (ou bien les méthodes de RUNGE-KUTTA-FEHLBERG). Son grand avantage est la stabilité à travers les discontinuités. On perd de la précision au début, mais la méthode peut quand même continuer. C'est pourquoi elle est idéale si les f ne sont pas des fonction analytiques mais des fonctions tabulées (résultats de mesures par exemple).

Les problèmes avec des constantes de temps très différentes posent des problèmes numériques si l'on veut appliquer les méthodes normales. GEAR a été

le premier a développer des schémas par différenciation rétrograde pour ces problèmes nommés « raides ». Les schémas permettent aussi un contrôle de précision par changement d'ordre et de pas.

Pour les problèmes pour lesquels on ne peut pas déterminer *a priori* leur « raideur » il faut trouver un algorithme qui peut changer le schéma en fonction des résultats numériques.

Bibliographie

[1] ACM: *Concurrent Object-Oriented Programming, Communications of the ACM*, Sep 1993. Special Issue.

[2] ALLAZ, CHRISTOPHE: *Développement d'un générateur expert de plan masse.* rapport de DEA, Stage scientifique ENPC, 1992.

[3] ANDERSON, JEFFREY L.: *A Network Definition and Solution of Simulation Problems.* Simulation Research Group, Lawrence Berkeley Laboratory, Berkeley, CA 94720, September 1986.

[4] BAASE, SARA: *Computer Algorithms.* Addison-Wesley Publishing Company, January 1983.

[5] BECHTOLD, M. et M. NIEMEYER: *Simulator Coupling System, flexible distributed simulation of electronic systems.* Rapport technique, cadlab, Paderborn, RFA, 1991.

[6] BECKER, JØRG D.: *Dynamik und Struktur in Systemen.* Dans : IATM [48].

[7] BLANC-SOMMEREUX, I., G. CAPLAIN, R. EBERT, B. FLAMENT, G. LEFEBVRE, A. NEVEU et B. PEUPORTIER: *Rapport AFME / ARMINES 1988/1989, Projet SYMBOL.* Rapport technique, CENERG -GISE, École des Mines de Paris, École Nationale des Ponts et Chaussées, 1990.

[8] BLANC-SOMMEREUX, ISABELLE: *Étude de couplage dynamique de composants de bâtiment par synthèse modale.* thèse de doctorat, École des Mines de Paris, 1991.

[9] BONIN, J.L., C. BUTTO, J.L. DUFRESNE, J.Y. GRANDPEIX, J.L. JOLY, A.LAHELLEC, V. PLATEL et M. RIGAL: *The ALMETH Project Zoom Code: Results and Perspectives.* Dans : CLARKE, J.A. et al. [24].

[10] BONIN, J. L., J. Y. GRANDPEIX, A. LELHASHAORI, J. L. JOLY et A. LAHELLEC: *Couplic Analysis in Building Thermal Simulation the Zoom program.* Dans : ., 1989.

[11] BOOCH, G.: *Software Components with Ada.* The Benjamin/Cummings Publishing Company, Inc., Menlo Park, USA, 2nd. tirage, 1986.

[12] BOOCH, GRADY: *Ingénierie du logiciel avec Ada.* InterEditions, Paris, 1988.

[13] BOUIA, H. et P. DALICIEUX: *Simplified Modelling of Air Movements Inside Dwelling Room.* Dans : CLARKE, J.A. et al. [24].

[14] BRAUN, M.: *Differential Equations and Their Applications.* Springer-Verlag, New York, 3rd tirage, 1986.

[15] BRING, AXEL: *IDA SOLVER, User's Documentation.* Department of Building Services Engineering, KTH, Stockholm, Suede, Jan 1992.

[16] BUHL, FRED, ENDER ERDEM, JEAN-MICHEL NATAF, FREDERICK C. WINKELMANN, MICHAEL A. MOSHIER et EDWARD F. SOWELL: *The U.S. EKS: Advances in the SPANK-based Energy Kernel System.* Dans : THE LABORATORY OF THERMODYNAMICS OF THE UNIVERSITY OF LIÈGE [76].

[17] CAI, WEN: *Dveloppement et applications de modèles d'échanges radiatifs par suivi de rayons.* thèse de doctorat, École des Mines de Paris, 1992.

[18] CAMP, ACSL /: *a Simulation Program and a Bondgraph Preprocessor for VAX and IBM PC*, 1988.

[19] CHAR, B. W., K. O. GEDDES et G. H. GONNET: *MAPLE User's Guide.* WATCOM Publications, Ltd., Waterloo, Ontario, Canada, 1985.

[20] CHURCHMAN, C. WEST: *The Systems Approach and Its Enemies.* Basic Books Inc., New York, 1979.

[21] CISI INGÉNIÈRIE, 35, bd Brune, 75680 Paris Cedex 14: *Note technique Neptunix, Phase numérique: manuel de l'utilisateur*, Sep 1983.

[22] CISI INGÉNIÈRIE, 3, rue Lecorbusier - Silic 232, 94528 Rungis Cedex: *NEPTUNIX: Langage de description de modèle*, Oct 1988.

[23] CISI INGÉNIÈRIE: *ALLAN. Manuel de référence*, 1992.

[24] CLARKE, J.A., J. W. MITCHELL et R.C. VAN DE PERRE (éditeurs): *Building Simulation '91*, Sophia-Antipolis, Nice, France, Aug 1991. IBPSA.

[25] COLLATZ, L.: *The Numerical Treatment of Differential Equations.* Springer-Verlag, New York, 1960.

[26] CROUZEIX, M. et A. L. MIGNOT: *Analyse numerique des équations différentielles.* Masson, Paris, 1984. très theorique, mais complet; ne traite des problèmes différentiels qu'à condition initiale.

[27] DAENZER, W.: *Systems Engineering.* Verlag Industrielle Organisation, Zurich, 1987.

[28] DUBOIS, A.-M.: *Component Model Documentation and Management: The Evolution of the Proforma and Modelotheque.* Dans : THE LABORATORY OF THERMODYNAMICS OF THE UNIVERSITY OF LIÈGE [76].

[29] DUBOIS, A.M., J.L. DUFRESNE, R. EBERT, J.Y. GRANDPEIX, J.L. JOLY, A.LAHELLEC, L.LARET, G.LEFEBVRE, J.L. PLAZY et M.POTTIER: *The Model Coupling Problem: Methods Used in Some Building Analysis Tools and the ALMETH Propositions.* Dans : CLARKE, J.A. et al. [24].

[30] EBERT, ROLF: *New Modular and Structures Approach to the Study of Complex Thermal Systems.* rapport de DEA, Technische Universitat Munchen, Lehrstuhl A fur Thermodynamik, 1989.

[31] EBERT, ROLF: *Essai de structurer quelques idées sur des mots mal compris.* papier interne de GISE, EMP-ENPC, janvier 1990.

[32] ENGELN-MULLGES, G. et F. REUTER: *Numerische Mathematik fur Ingenieure.* Wissenschaftsverlag, Mannheim, 5th tirage, 1987.

[33] ENRIGHT, W. H. et T. E. HULL: *Test Results for Non Stiff Ordinary Differential Equations.* SIAM J. Num. Anal., 13:944–950, 1976.

[34] ETIENNE WURTZ, ROLF EBERT, GILLES LEFEBVRE ET JEAN-MICHEL NATAF: *Couplage Conduction-Convection-Rayonnement en environnement orienté objet.* Dans : *Colloque Fanco-Quebecois.* Université Paul-Sabatier, 1993.

[35] FANSIM: *Program for Fourier and Modal Analysis.* Meerman Automation, NL-7160 AC Neede, Postbus 154, Oct 1988.

[36] FLAMENT, BERNARD: *Synthèse modale de systèmes thermiques.* thèse de doctorat, École des Mines de Paris, 1993.

[37] FRANCHISSEUR, R.: *Comparaison de Zoom et Neptunix.* communication directe, 1992.

[38] FRANCHISSEUR, R. et J. M. NATAF: *Comparaison de Zoom et Spark sur la cellule de Hambourg.* communication directe, 1993.

[39] GARBOW, B. S., K. E. HILLSTROM et J. J. MORE: *The MINPACK Project.* Rapport technique, Argonne National Laboratory, March 1980.

[40] GAUTIER, B., F. X. RONGÈRE et D. BONNEAU: *CLIM 2000: The Building Energy Simulation Tool and the Modelling Method.* Dans : CLARKE, J.A. et al. [24].

[41] GEAR, C. W.: *Numerical Initial Value Problems in Ordinary Differential Equations.* Prentice-Hall, N.J., 1971.

[42] GIQUEL, RENAUD: *Approche moderne des systèmes énergétiques.* SFT, Oct. 1992.

[43] GOMEZ, PETER et GILBERT J. B. PROBST: *Verneztes Denken im Management.* Die Orientierung. Schweitzer Volksbank, Berne, Suisse, 1987.

[44] HENRICI, P.: *Discrete Variable Methods in Ordinary Differential Equations.* John Wiley, New York, 1962. grand classique pour les calculs des équations différentielles.

[45] HERRARTE, VIRGINIA et EWING LUSK: *Studying Parallel Program Behaviour with* `Upshot`. Argonne National Laboratory.

[46] HEYDEMANN, M.: *ASTEC3, Manuel de référence - utilisateur.* CISI, 1981.

[47] HULL, T. E., W. H. ENRIGHT, FELLEN et SEDGWICK: *Comparing Numerical Methods for Ordinary Differential Equations.* SIAM J. Num. Anal., 9(4):603–637, 1972.

[48] IATM (éditeur): *Systemansatz in Theorie und Praxis*, TU München, 1991.

[49] JEANDEL, ALEXANDRE et ISABELLE PALERO: *Thermal Modelling And Simulation of Buildings at Gaz de France.* Dans : THE LABORATORY OF THERMODYNAMICS OF THE UNIVERSITY OF LIÈGE [76].

[50] KROY, W.: *Systemansatz in der Technik.* Dans : IATM [48].

[51] LEFEBVRE, G.: *Projet SYMBOL-2, rapport final convention AFME.* Rapport technique, GISE, École des Mines de Paris, École Nationale des Ponts et Chaussées, Juil. 1991.

[52] LEFEBVRE, GILLES: *Analyse et réduction modales d'un modèle de comportement thermique de bâtiment.* thèse de doctorat, École des Mines de Paris, Paris, 1987.

[53] LEMOIGNE, JEAN-LOUIS: *La théorie du système général, Théorie de la modélisation.* Presses Universitaires de France, Paris, 3. tirage, 1990.

[54] LEVY, H. et D. W. LOW: *A new algorithm for finding small cycle cut sets.* IBM Scientific Center, Los Angeles, June 1983.

[55] LE, XIAOHUA: *Implémentation d'un modèle d'acteur, Application au traitement de données partielles en audit thermique de bâtiment.* thèse de doctorat, École Nationale des Ponts et Chaussées, 1992.

[56] MAYINGER, F.: *Systemtechnik und systemtechnische Anwendungen.* notes de cours de la Technische Universitat München, 1989/1990.

[57] MEERMAN AUTOMATION, NL-7160 AC Neede, Postbus 154: *TUTSIM on IBM PC Computer, Users Manual*, Oct 1990.

[58] MEYER, B.: *Objet-Oriented Software Construction.* Prentice Hall International (UK) Ltd., Hertfordshire, Great Britain, 1988.

[59] MIT: *MACSYMA Reference Manual, version 10.* Cambridge, MA, 1983.

[60] MORIN, EDGAR: *La méthode, La nature de la nature.* Éditions de Seuil, Paris, 1977.

[61] NATAF, JEAN-MICHEL et ROLF EBERT: *On the Efficiency of Partitioning in Object-Based Simulation.* Dans : *Building Simulation '93*, Adelaide, Aug 1993.

[62] OULEFKI, ABDELHAKIM: *Réduction de modèles thermiques par amalgame modale.* thèse de doctorat, École Nationale des Ponts et Chaussées, 1993.

[63] PETZOLD, LINDA R.: *A Description of Dassl: a Differential/Algebraic System Solver.* Rapport technique, Sandia National Laboratories, 1982.

[64] POWELL, M. J. D.: *A Hybrid Method for Nonlinear Equations, Numerical Methods for Nonlinear Algebraic Equations.* Gordon and Breach, 1970.

[65] PRATT, DAVID B., PHILLIP A. FARRINGTON, CHUDA B. BASNET, HEMANT C. BHUSKUTE, MANJUNATH KAMATH et JOE H. MIZE: *A Framework for Highly Reusable Simulation Modeling: Separating Physical, Information, and Control Elements.* Dans : RUTAN, ALAN H. (éditeur): *The 24th Simulation Symposium, Record of Proceedings*, New Orleans. Louisiana, April 1991. IEEE Computer Society Press.

[66] PRESS, W. H., B. P. FLANNERY, S. A. TEUKOLSKY et W. T VEHERLING: *Numerical Receipes in C, The Art of Scientific Computing.* Cambridge University Press, Cambridge, 1988. un point de vue très pratique pour presque tous les problèmes numeriques.

[67] RIEDIJK, W.: *Appropriate Technology for Developping Countries.* Delft University Press, Delft, Pays-Bas, third tirage, 1987.

[68] RONGÈRE, F. X.: *What is an Elementary Model?* Rapport technique, Electricité de France, 1991. distribué pour la réunion ALMETH.

[69] ROSENBERG, R.: *A Users Guide to ENPORT-4.* Wiley, New York, 1974.

[70] SAHLIN, P.: *IDA, a Modelling and Simulation Environment for Building Applications.* Rapport technique, ITM Swedish Institut of Applied Mathematics, Teknikpark, S-41288 Geteborg, Dec 1991.

[71] SAHLIN, PER et AXEL BRING: *IDA SOLVER, A Tool for Building and Energy Systems Simulation.* Dans : CLARKE, J.A. et al. [24].

[72] SCHEIDT, JURGEN VON: *Konzepte fur die Zukunft, das neue Denken in Wissenschaft und Wirtschaft.* Bonn Aktuell Verlag, Stuttgart, Munchen, 1991.

[73] SOLAR ENERGY LABORATORY, UNIVERSITY OF WISCONSIN - MADISON: *TRNSYS, a Transient Simulation Program*, April 1984.

[74] SOWELL, E. F. et P. SAHLIN: *Neutral Format and Automatic Translations for Building Simulation Submodels.* Dans : IBPSA (éditeur): *Building Simulation '89*, Vancouver, B. C., Canada, Juin 1991.

[75] STOER, J. et R. BULIRSCH: *Introduction to Numerical Analysis.* Springer-Verlag, New York, second tirage, 1980.

[76] THE LABORATORY OF THERMODYNAMICS OF THE UNIVERSITY OF LIÈGE (éditeur): *3rd International Conference on System Simulation in Buildings*, Liège, Belgium, Dec 1991.

[77] THOMA, JEAN U.: *Simulation by Bondgraphs : Introduction to a Graphical Method.* Springer-Verlag, Berlin Heidelberg New York, 1990.

[78] VESTER, FREDERIK: *Systemstudie Ökoland*, tome I - VI. Universitat der Bundeswehr, Institut fur Interdependenz von Technik und Gesellschaft, 1983 - 1988.

[79] VESTER, FREDERIK: *Welche Rolle spielt das Auto morgen?* Institut fur Interdependenz von Technik und Gesellschaft, Universitat der Bundeswehr, Munchen, 1983.

[80] WURTZ, ÉTIENNE: *IRADI, un logiciel pour le comfort radiatif.* papier interne de GISE, EMP - ENPC, 1993.

[81] ZEIGLER, B. P.: *Theory of Modelling and Simulation.* Krieger Publishing Company, Malabar, Florida, 1976.

[82] ZOOM, COLLABORATION: *FET détaillé*, Oct 1991. AFME - (GER ALMETH).

[83] ZOOM, COLLABORATION: *Introduction au FET, découpage - raccordement*, Oct 1991. AFME - (GER ALMETH).

[84] ZOOM, COLLABORATION: *Introduction au FET, exemples simples*, Oct 1991. AFME - (GER ALMETH).

[85] ZOOM, COLLABORATION: *Quelques exemples et leurs processeurs*, Oct 1991. AFME - (GER ALMETH).

[86] ZOOM, COLLABORATION: *séminaire TEF - ZOOM.* AFME - (GER ALMETH), Oct 1991.